UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Sistemas Informáticos y Computación

TESIS DOCTORAL

Aplicación de las Unidades de Procesamiento Gráfico en el diseño e implementación de sistemas de ray tracing

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Roberto Torres de Alba

Directores

Pedro Jesús Martín de la Calle
Antonio Gavilanes Franco

Madrid, 2014

Aplicación de las Unidades de Procesamiento Gráfico en el diseño e implementación de sistemas de ray tracing

Roberto Torres de Alba

Tesis Doctoral

Directores:
Pedro Jesús Martín de la Calle
Antonio Gavilanes Franco

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid
Diciembre de 2013

Aplicación de las Unidades de Procesamiento Gráfico en el diseño e implementación de sistemas de ray tracing

Roberto Torres de Alba

Tesis doctoral en formato publicaciones presentada por el doctorando Roberto Torres de Alba en el departamento de *Sistemas Informáticos y Computación* de la *Universidad Complutense de Madrid* para la obtención del título de doctor en Ingeniería Informática.

Terminada el día 15 de diciembre de 2013.

Título:

Aplicación de las Unidades de Procesamiento Gráfico en el diseño e implementación de sistemas de ray tracing

Autor:

Roberto Torres de Alba (r.torres@fdi.ucm.es)

Directores:

Pedro Jesús Martín de la Calle (pjmartin@sip.ucm.es)
Antonio Gavilanes Franco (agav@sip.ucm.es)

Agradecimientos

Esta tesis no hubiera sido posible sin la ayuda y la colaboración de mucha gente, a los que me gustaría agradecer en los siguientes párrafos.

Primeramente, a mis directores de tesis, Chus y Antonio, cuya ayuda ha sido imprescindible para la realización de esta tesis. Sobre todo, por las reuniones científicas que hemos tenido, algunas alargándose incluso hasta la noche. También por su ayuda y sus consejos para la mejora de los artículos asociados con esta tesis, en la mejora de este documento, así como por la ayuda con la documentación necesaria.

En segundo lugar, a mi familia por aguantarme y por darme apoyo y cariño durante el tiempo de realización de esta tesis.

Me gustaría también agradecer a las siguientes personas: a Ana Gil por ser la directora de mi trabajo del DEA, que fue el precursor de esta tesis sobre GPUs; a los revisores por sus comentarios, que han contribuido a mejorar tanto nuestros artículos como el documento de esta tesis; a mis compañeros de la *Facultad de Informática* de la *Universidad Complutense de Madrid*, especialmente a los habitantes de los despachos 219 y 220.

Los resultados experimentales de estos trabajos no se hubieran podido realizar sin las escenas usadas, a cuyos creadores me gustaría agradecer. Así, agradezco a *The Stanford 3D Scanning Repository*[1] por las escenas Bunny, Dragon y Happy Buddha; a *The Utah 3D Animation Repository*[2] por las escenas FairyForest, Runner y Toys; a Anat Grynberg y Greg Ward por la escena ConfRoom; a Marko Dabrovic por las escenas Sponza y Sibenik; y finalmente al *Taller Multimedia* de la *Universidad Complutense de Madrid* por la escena Skull.

También agradezco a los proyectos *Análisis y Aprovechamiento de la Coherencia de Rayos para un Ray Tracer* (CCG10-UCM/TIC-5476), *Aplicación de CUDA en la Síntesis de Imágenes Realistas* (CCG08-UCM/TIC-4252), *Programa de Financiación de Grupos de Investigación* (GR35/10-A-921547) y *Programa de Creación y Consolidación de Grupos de Investigación* (GR58/08-C-921547), que han financiado a nuestro grupo de investigación *Grupo de Informática Gráfica*. Estos proyectos han sido financiados por las entidades Universidad Complutense de Madrid, Comunidad de Madrid, Banco Santander y Banco Santander Central Hispano.

[1] `http://graphics.stanford.edu/data/3Dscanrep/`

[2] `http://www.sci.utah.edu/~wald/animrep/`

Resumen

La obtención de imágenes que sean indistinguibles de una fotografía es uno de los objetivos de la informática gráfica. Los algoritmos de *ray tracing* son los que obtienen las imágenes con mayor calidad. Estos algoritmos tienen en común que simulan el comportamiento de la luz trazando rayos a través de una escena tridimensional. Sin embargo, se tratan de algoritmos lentos debido a la gran cantidad de rayos que se necesitan trazar para renderizar imágenes con buena calidad.

Una operación común en los algoritmos de ray tracing consiste en encontrar el primer objeto de la escena intersecado por cada rayo. Para realizar esta tarea más eficientemente, se han desarrollado estructuras de datos, generalmente jerárquicas, que organizan la escena. Estas estructuras reciben el nombre de *estructuras de aceleración*. Por lo tanto, una fase importante de los algoritmos de ray tracing consiste en el *recorrido* de los rayos por la estructura de aceleración de la escena.

Las *Unidades de Procesamiento Gráfico* (o *GPUs*) son un hardware que fue inicialmente diseñado para implementar el algoritmo de la *tubería gráfica*. Con el tiempo, han evolucionado hasta convertirse en la actualidad en un hardware paralelo que es completamente programable. Debido a su naturaleza paralela, la potencia de cálculo de las GPUs ha llegado a superar, a día de hoy, a la de las CPUs. Este hecho, unido al relativamente bajo coste de las GPUs, las ha convertido en un hardware ampliamente difundido.

Los algoritmos de ray tracing son altamente paralelos debido a que los rayos trazados son independientes entre sí. Este hecho es la principal razón para usar las GPUs como hardware sobre el que implementar el ray tracing. Desafortunadamente, una implementación directa no saca el máximo partido de las GPUs. Las GPUs están formadas por una serie de unidades funcionales que se ejecutan en paralelo. Para que estas unidades no se queden paradas, necesitan que se les proporcione datos sobre los que operar. Sin embargo, su memoria *off-chip* no proporciona estos datos con suficiente velocidad, por lo que se necesita una modificación del recorrido de los rayos a través de la estructura de aceleración. En esta tesis proponemos tres modificaciones de este algoritmo.

En primer lugar, hemos implementado un algoritmo de recorrido sin pila de una BVH hilvanada. En este recorrido, grupos de rayos, llamados *paquetes*, recorren juntos la estructura de aceleración. Los rayos de cada paquete colaboran cada vez que leen datos de memoria, lo que aumenta la eficiencia de estos accesos. En segundo lugar, hemos desarrollado un nuevo algoritmo de recorrido de una estructura de aceleración jerárquica. El objetivo de este recorrido consiste en mejorar el rendimiento de los rayos que son geométricamente muy diferentes, por medio de un recorrido secuencial de ciertos subárboles de la estructura. En tercer lugar, hemos modificado la forma en que se generan los rayos para que las lecturas de memoria sean más eficientes. Esta nueva generación permite trazar más rayos en el mismo tiempo, mejorando la calidad de las imágenes resultantes.

La forma en que se construye la estructura de aceleración a partir de la escena influye en el número de nodos que necesita recorrer cada rayo antes de encontrar su objeto intersecado más cercano, lo que determina el rendimiento global de la aplicación. El algoritmo de construcción usado actualmente está basado en una heurística conocida como *heurística del área de la superficie*

(o *SAH*). Nosotros proponemos varias modificaciones de esta heurística que se especializa sobre un conjunto de rayos cuyas direcciones se encuentran restringidas. Esta especialización mejora la eficiencia de la estructura cuando los rayos exploran la escena.

En lo que respecta a la programación de las GPUs, hemos realizado una adaptación a este hardware del algoritmo de Dijkstra que resuelve el problema del camino más corto desde un nodo hasta todos los demás. Este algoritmo elige un nodo de entre aquellos que tienen su menor estimación del camino más corto. Este nodo elegido será posteriormente usado para actualizar las estimaciones de sus nodos adyacentes. El esquema secuencial del algoritmo tradicional de Dijkstra no se adapta adecuadamente al hardware paralelo de las GPUs. Nosotros hemos analizado el rendimiento de un algoritmo paralelo consistente en manejar paralelamente todos los nodos que tienen la misma estimación mínima. Este algoritmo paralelo obtiene mejor rendimiento que el clásico secuencial en CPU.

Desde el punto de vista del programador, una aplicación se puede descomponer en una serie de operaciones sencillas, conocidas como *primitivas*. En el ámbito de la programación paralela, estas operaciones reciben el nombre de *primitivas de datos paralelos*, y algunas ya han sido implementadas en GPU. Ya que estas operaciones son la base de muchas aplicaciones, el rendimiento general del sistema depende en gran medida del rendimiento de estas primitivas.

Una de estas primitivas es la llamada primitiva de *reducción*. En su versión *no segmentada*, consiste en aplicar una operación binaria sobre todos los elementos del array de entrada. En su versión *segmentada*, el array de entrada se encuentra dividido en secciones contiguas, de forma que la reducción segmentada se encarga de reducir cada una de estas secciones por separado. Nosotros hemos analizado el comportamiento en GPU de tres implementaciones de esta primitiva en sus versiones segmentada y no segmentada. Hemos llegado a la conclusión de que no existe una implementación que obtenga el mejor rendimiento en cualquier situación para la versión segmentada, ya que depende de la distribución de los segmentos en la entrada. Además, hemos propuesto y probado dos optimizaciones que aceleran el rendimiento del mejor algoritmo conocido hasta entonces para la versión no segmentada.

Organización de la Tesis

Esta tesis está organizada de la siguiente manera.

Capítulo 1. En este capítulo presentamos el modelo de la luz que usaremos a lo largo de la tesis, junto con la forma en que la luz interactúa con los objetos de la escena. En concreto, consideramos la luz como partículas que viajan rápidamente en línea recta y que pueden ser absorbidas o reflejadas en la superficie de los objetos. Este modelo nos sirve para definir las ecuaciones que determinan los colores de los píxeles de las imágenes renderizadas. Estas ecuaciones, que tienen la forma de integrales, se aproximarán por medio de los métodos de Monte Carlo. Estos métodos consisten en trazar rutas aleatorias entre las luces de la escena y la cámara, y mediar sus contribuciones.

Capítulo 2. En este capítulo presentamos la arquitectura de las *Unidades de Procesamiento Gráfico* (*GPU*), el hardware paralelo que se usará a lo largo de esta tesis. También se describe la operación *scan*, o suma de prefijos, que realiza la suma de todos los elementos anteriores a cada componente de un array. La operación scan es la base sobre la que se implementan otras operaciones paralelas, llamadas *primitivas*. En la sección 2.5, presentamos nuestro trabajo [MATG12], consistente en un análisis de diversas implementaciones en GPU de la primitiva *reducción*. Esta primitiva consiste en la obtención de un único valor a partir de un array al que se le ha aplicado una operación binaria sobre todos sus elementos. Además, presentamos también dos optimizaciones algorítmicas que mejoran su rendimiento. En la sección 2.6 presentamos nuestros trabajos [MTG09] y [MTG08], consistentes en una adaptación a GPU del algoritmo de Dijkstra que encuentra el camino más corto desde un nodo de un grafo hasta todos los demás.

Capítulo 3. En este capítulo presentamos las estructuras de datos encargadas de organizar la escena para hacer más eficientes las consultas de los rayos. Presentamos también la heurística del área de la superficie (SAH), usada para construir estructuras eficientes. En la sección 3.9 presentamos nuestros trabajos [TMGA12] y [TMG09b], que consisten en una modificación de la SAH para construir estructuras especializadas en un conjunto de rayos con direcciones restringidas.

Capítulo 4. En este capítulo se presentan diversas técnicas para aprovechar el hardware sobre el que se ejecutan los algoritmos de ray tracing. Además, se presenta el algoritmo de recorrido en GPU más usado en esta tesis. En la sección 4.5 presentamos nuestros trabajos [TMG09a] y [TMG08], consistentes en implementar, en GPU, un recorrido sin pila de una BVH cuyos nodos se han ampliado con hilvanes. En la sección 4.6 presentamos nuestro trabajo [TMG11], consistente en una nueva forma de recorrer una BVH en la que ciertos nodos se seleccionan para realizar un filtrado de los rayos, mejorando la eficiencia de un conjunto de rayos muy incoherentes.

Capítulo 5. En este capítulo presentamos nuestro trabajo [TMG12], consistente en una manera alternativa de generar rutas aleatorias. Esta nueva forma de generar rutas permite una mejor utilización de los recursos de las GPUs, lo que permite trazar más rutas en el mismo intervalo de tiempo, mejorando consecuentemente la calidad de las imágenes renderizadas.

Índice general

Parte I

Descripción de la Investigación

Capítulo 1

Modelo Computacional de la Luz y la Materia

1.1. Introducción

Uno de los objetivos de la *Informática Gráfica* es que un computador sea capaz de realizar imágenes subjetivamente indistinguibles de una fotografía. Para conseguir este tipo de imágenes es necesario definir un modelo computacional que describa el comportamiento de la luz en el mundo físico. Para ello, tenemos que determinar cómo están definidos los *objetos* tridimensionales de nuestra *escena* y cómo la luz interactúa con ellos.

Para nuestros propósitos, una escena está formada por una lista de objetos tridimensionales. Un objeto está definido por cualquier figura geométrica para la que se pueda calcular su intersección con un *rayo* (la justificación de este hecho se verá en la sección 1.5). Un rayo está definido por su *origen* o y su *dirección* d, de forma que el origen es un punto en el espacio $\mathbb{R}^3$ y la dirección es un vector unitario. Por convenio, la dirección se suele definir como el vector que señala un punto sobre la superficie de la esfera unidad centrada en el origen, a la que denotaremos por $\mathcal{S}^2$. Estos dos elementos juntos forman una semirrecta llamada *rayo*

$$rayo(o, d) = \left\{ o + td \mid t \in \mathbb{R}^+ \right\}$$

Además de la existencia de la intersección rayo-objeto, también es necesario que la función de intersección devuelva un vector *normal* a la superficie del objeto, que se usará para los cálculos de iluminación y para determinar la parte exterior e interior de los objetos. Por convenio, la parte a la que apunta el vector normal se considera la parte *exterior*, mientras que la otra se considera el *interior* del objeto. Denotaremos por $\mathcal{M}$ el conjunto de todos los puntos que se encuentren sobre la superficie de los objetos de la escena.

Para nuestros propósitos es suficiente con que los objetos estén formados por *mallas de triángulos*. Los triángulos son de uso común en Informática Gráfica ya que son capaces de aproximar casi cualquier objeto. Los *vértices* de cada triángulo pueden considerarse puntos reales de los objetos que se desean aproximar, por lo que es habitual asociarles una normal a cada uno. Las normales sobre la superficie de los triángulos se obtienen interpolando las normales de sus vértices usando sus *coordenadas baricéntricas*. Esto permite dar una apariencia curvada a los triángulos, a pesar de que realmente sus superficies son planas.

Aparte de la información sobre las normales, añadiremos también información sobre las propiedades físicas a cada objeto (sección 1.4). Estas propiedades indican si los objetos emiten luz[1]

[1]Si un objeto emite luz recibe el nombre de *luminaria* (o *luminaire* en inglés).

y cómo se comporta la luz cuando llega a su superficie. Esto permite simular el color y el material de los objetos a fin de imitar la apariencia que tendrían en el mundo real.

Para formar la imagen final, es necesario un dispositivo que recoja y guarde la información de la luz que le llega. A ese dispositivo le llamamos *cámara*. La cámara recoge la luz que recibe de los objetos de la escena, ya sea directamente, si el objeto emite luz, o indirectamente a través de reflexiones entre otros objetos de la escena. Dentro de la cámara se encuentra un plano, llamado *película*, donde se encuentra una matriz de receptores de luz, llamados *píxeles*, que reaccionan a la luz entrante. La información contenida en esos receptores es lo que forma la imagen final. El proceso completo de obtención de dichos valores se conoce como *renderizado*. La cámara que usaremos sigue un esquema semejante al de una cámara oscura o *pinhole*, es decir, que la luz solo entra en la cámara a través de un pequeño agujero.

1.2. Modelo de Partículas de la Luz

Para generar imágenes realistas no es necesario simular el comportamiento de la luz según el modelo actual de la física[2]. Un modelo más sencillo, basado en *energía radiante*, es suficiente para conseguir un alto grado de realismo en las imágenes generadas por computador[3]. Desgraciadamente, este modelo excluye ciertos fenómenos que produce la luz en el mundo real tales como *polarización*, *interferencia* o *fosforescencia*. Sin embargo, estos fenómenos o no son muy significativos o pueden ser simulados de otra manera con este modelo.

Vamos a suponer que la luz está formada por *partículas puntuales*[4] que viajan en línea recta a la misma velocidad y que no interactúan entre sí. Cada una de estas partículas está caracterizada por un estado consistente en su *posición* en el espacio, su vector *velocidad* y su *longitud de onda*[5]. Estas partículas se generan en la superficie de las luminarias y viajan en línea recta en el vacío hasta que encuentran la superficie de otro objeto. En ese momento puede ocurrir que la partícula sea *absorbida* por el objeto, en cuyo caso desaparece y no contribuye más al renderizado de la imagen final. Por el contrario, es posible que la partícula sea lanzada nuevamente hacia el exterior, viajando otra vez en línea recta hasta que se encuentre con otro objeto. A este último caso lo llamaremos *difusión* (en inglés, *scattering*). En el momento de la difusión, solo cambia la dirección en la que viaja la partícula, manteniéndose igual el resto de su estado. Así, su posición, el módulo de su velocidad y su longitud de onda permanecen inalteradas antes y después del choque. La nueva dirección depende de las propiedades del material del objeto y suele ser aleatoria dentro de un rango de direcciones.

Durante la vida de las partículas, es posible que algunas lleguen hasta la cámara, siendo detectadas por uno o varios píxeles de la película y, por lo tanto, contribuyendo a la imagen final. Asumimos que la cámara no modifica el comportamiento de las partículas que la atraviesan, por lo que en la imagen final no aparece ningún indicio de la presencia de la propia cámara.

1.3. Medición de las Partículas

Para derivar las magnitudes que usaremos durante el renderizado tenemos que medir la cantidad de partículas que se encuentran en un determinado volumen y en un determinado intervalo

[2] Una introducción divulgativa del modelo de las partículas en la Física se puede encontrar en el libro de Richard Feynman "Electrodinámica Cuántica: La extraña teoría de la luz y la materia" [Fey89].

[3] El modelo de partículas que se describe en esta sección está basado en el Capítulo 2 de la tesis de J. Arvo [Arv95] y en la tesis de E. Veach [Vea98].

[4] Se podría llamar a estas partículas *fotones* aunque, debido a que no tienen relación con los fotones físicos y que pueden confundirse con los fotones del *photon mapping*, se ha preferido no hacerlo.

[5] La longitud de onda de una partícula determina su color. De aquí en adelante vamos a suponer que solo existen tres colores posibles, *rojo*, *verde* y *azul*, que corresponden con el modelo *RGB*.

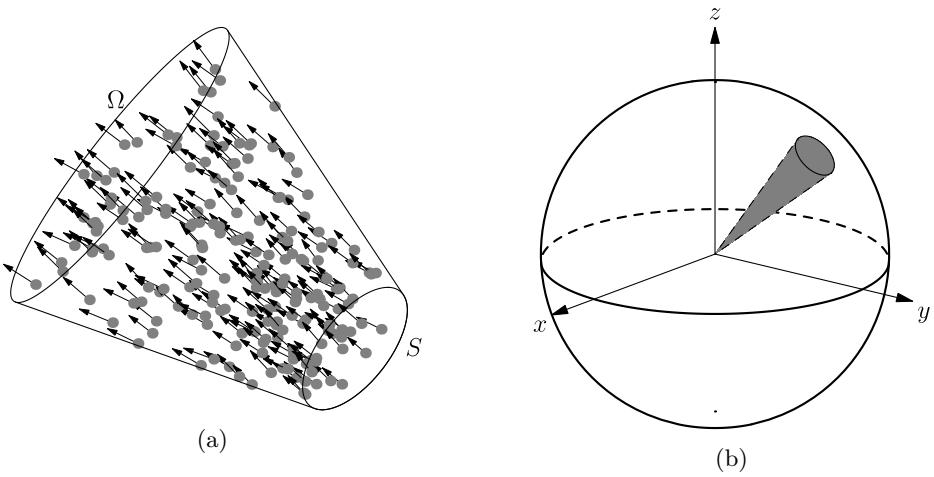


Figura 1.1: Fig. (a). Un conjunto de partículas que pasa por una superficie S y cuyas direcciones pertenecen a Ω forman un cono en un determinado intervalo de tiempo $[t_0, t_1]$. Fig. (b). Un ángulo sólido es el área de una sección de la esfera de radio unidad $\mathcal{S}^2$.

de tiempo. Una forma de hacerlo sería contar cuántas partículas se encuentran dentro del volumen. Sin embargo, asumiremos que el número de partículas emitidas desde las fuentes de luz es tan alto que se puede justificar el uso del cálculo matemático para medirlas. Concretamente, derivaremos la *radiancia* (o *energía radiante*) como magnitud fundamental, que nos servirá para derivar las demás.

Un conjunto de partículas en movimiento definen un volumen en el espacio en un intervalo de tiempo. Consideremos el conjunto de todas las partículas que atraviesan una cierta superficie S desde su parte interior a la exterior. Además, restrinjamos el conjunto de partículas anterior a solo aquellas cuyas direcciones se encuentra dentro de un conjunto de direcciones Ω. En un cierto intervalo de tiempo $[t_0, t_1]$, el conjunto de partículas con las condiciones anteriores forman un volumen similar a un cono (figura 1.1a).

Para medir ese volumen vamos a usar medidas usuales sobre superficies, direcciones y tiempo. Como medida sobre superficies usaremos el *área*, denotada como A y medida en *metros cuadrados* $[m^2]$. Para el tiempo t usamos la longitud del intervalo, medido en *segundos* $[s]$. Las direcciones, al ser puntos sobre la superficie de una esfera unidad, se miden con el ángulo sólido que forman, denotado como σ. Un ángulo sólido (figura 1.1b) se define como el área de una sección de la superficie de la esfera de radio unidad[6]. No tiene unidades de medida, aunque se suele expresar con *esterorradianes* $[sr]$.

Por lo tanto, definimos la medida η como la "cantidad" de partículas que pasan a través de la superficie S con direcciones en Ω en el intervalo de tiempo $[t_0, t_1]$ como

$$\eta(S \to \Omega, [t_0, t_1]) = \int_{t_0}^{t_1} \int_{\omega \in \Omega} \int_{x \in S} 1 \; dA(x) \, d\sigma(\omega) \, dt$$

La flecha $\to$ indica que las partículas atraviesan la superficie desde la parte interior hacia la exterior. Simétricamente, usaremos la flecha $\leftarrow$ si las partículas llegan a la superficie. En caso de

[6]Un *ángulo* en dos dimensiones se define como una sección de circunferencia dividido entre su radio. De manera semejante, un *ángulo sólido* en tres dimensiones se define como el área de una sección de circunferencia dividido entre el radio al cuadrado. Ambas relaciones se mantienen constantes independientemente de cuáles sean los radios elegidos. En este cociente se aprecia el carácter adimensional de ambas medidas.

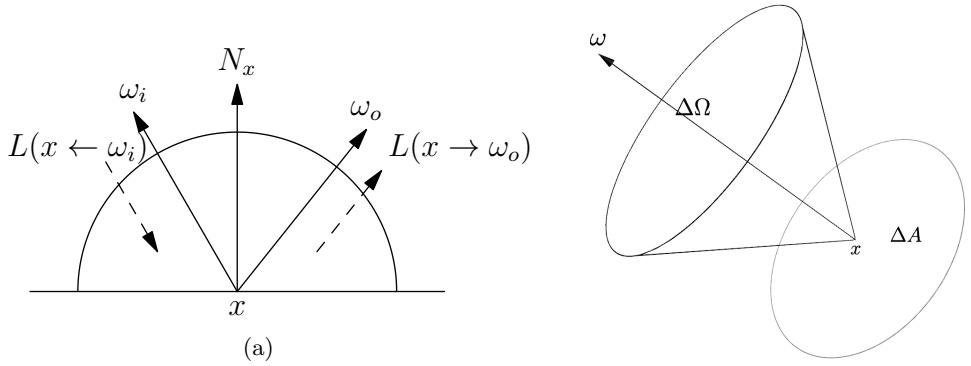


Figura 1.2: Fig. (a). Radiancia entrante y saliente en el punto x de una superficie por las direcciones ω_i y ω_o, respectivamente. Las flechas discontinuas muestran la dirección de la radiancia. Como se aprecia, el vector de dirección ω_i apunta en la dirección contraria al movimiento de las partículas. Fig. (b). Si se estrecha $\Delta\Omega$ sobre ω y ΔA sobre x, la relación $\frac{\Delta\Phi}{\Delta A \Delta\Omega}$ tiende a la radiancia $L(x \leftarrow \omega)$.

que las partículas lleguen a la superficie, las direcciones de Ω se consideran que son contrarias a las direcciones de las partículas (figura 1.2a). Si no existe ninguna dependencia entre los tres conjuntos S, Ω y $[t_0, t_1]$, entonces la medida de las partículas será simplemente el producto de las medidas de cada conjunto

$$\eta(S \rightarrow \Omega, [t_0, t_1]) = A(S)\,\sigma(\Omega)\,(t_1 - t_0)$$

1.3.1. Derivación de la Radiancia

La medida más común para un conjunto de partículas es la *energía* que llevan, a la que designaremos como Q, medida en *Julios* $[J]$. Vamos a asumir que Q y η son *absolutamente continuas* (denotado como $Q \ll \eta$), es decir, que cualquier conjunto de partículas con medida cero ($\eta = 0$) no lleva energía ($Q = 0$). Esto parece lógico ya que si un volumen tiene medida cero, entonces no contendrá ninguna partícula, por lo que no llevará nada de energía.

Al ser estas medidas absolutamente continuas, se garantiza la existencia de una función de densidad que relaciona Q y η (ver [CK04], página 190). Esa función de densidad se le llama *radiancia* (*radiance* en inglés) y se la denota como L (Nicodemus [Nic63]). La función L relaciona las medidas Q y η de la siguiente manera

$$Q = \int L\, d\eta$$

Se suele emplear la notación de la derivada para designar a esta densidad

$$L(x \rightarrow \omega, t) = \frac{dQ}{d\eta} = \frac{dQ}{dA^{\omega}(x)\, d\sigma_x(\omega)\, dt} \tag{1.1}$$

donde $dA^{\omega}(x)$ destaca el hecho de que la normal en el punto x es la misma que la dirección ω, y σ_x destaca el hecho de que el ángulo sólido se mide situando el origen de la circunferencia $\mathcal{S}^2$ en el punto x. Las unidades de medida de la radiancia son $[J \cdot m^{-2} \cdot sr^{-1} \cdot s^{-1}]$.

Se define el *flujo de energía* Φ como la energía por unidad de tiempo

$$\Phi = \frac{dQ}{dt}$$

medida en vatios $[W = J \cdot s^{-1}]$. Se puede demostrar (Kajiya [Kaj86]) que, en un sistema como el que se ha descrito, el flujo de energía tiende al *equilibrio de flujo* a medida que pasa el tiempo. En un sistema en equilibrio, el flujo no depende del tiempo, sino solo de las superficies de los objetos y las direcciones. Ya que la velocidad de las partículas de la luz es muy alta, suponemos que se llega al equilibrio en un tiempo tan pequeño que es despreciable. Por tanto, asumimos que durante el renderizado de la imagen, la escena ya se encuentra en un sistema en equilibrio de flujo.

Como suele ser habitual en los libros de Informática Gráfica (por ejemplo, en Pharr y Humphreys [PH10]), la radiancia se expresa en función del flujo de luz y no a través de la energía. Por tanto, la ecuación 1.1 se puede escribir, de manera equivalente como

$$L(x \to \omega) = \frac{d\Phi}{dA^{\omega}(x)\, d\sigma_x(\omega)}$$

Así, la radiancia se define como el flujo de energía por unidad de área y de ángulo sólido, medido en $[W \cdot m^{-2} \cdot sr^{-1}]$.

Una forma intuitiva de entender el significado de la radiancia es la siguiente (Shirley y Morley [SM03], págs. 158–161). Consideremos un punto x sobre la superficie de un objeto. Sea S una sección del plano tangente en x de manera que $x \in S$ (figura 1.2b). Sea también ω el vector normal al plano tangente y Ω un conjunto de direcciones que contiene a ω. Una aproximación de la radiancia entrante en x por ω, $L(x \leftarrow \omega)$, se obtiene si se divide el flujo entrante, al que denotamos como $\Delta\Phi$, entre el producto de ΔA (área de S) y $\Delta\sigma$ (ángulo sólido de Ω)

$$L(x \leftarrow \omega) \approx \frac{\Delta\Phi}{\Delta A \Delta\sigma} \tag{1.2}$$

A medida que los conjuntos S y Ω se hacen más pequeños alrededor de x y ω, respectivamente, los valores $\Delta\Phi$, ΔA y $\Delta\sigma$ se hacen también más pequeños y la relación 1.2 aproxima mejor la radiancia. En el límite, la relación 1.2 se considera la radiancia

$$\lim_{\substack{\Delta A \to 0 \\ \Delta\sigma \to 0}} \frac{\Delta\Phi}{\Delta A \Delta\sigma} = \frac{d\Phi}{dA^{\omega}(x)\, d\sigma^{x}(\omega)} = L(x \leftarrow \omega)$$

Obsérvese que la restricción de que x se encuentre sobre la superficie de un objeto se puede eliminar. Así, es posible calcular la radiancia entrante o saliente en un punto cualquiera del espacio si se coloca un plano que pasa por ese punto y se evalúa la energía que llega o sale por su normal. Por tanto, asumiremos la existencia de la función de radiancia —entrante o saliente— en todo punto del espacio y en cualquier dirección.

La radiancia es la magnitud adecuada para medir un haz de partículas que viajan por una misma línea (*haz de luz*) ya que su valor se conserva en todos los puntos de dicha línea. Esta propiedad se expresa diciendo que la radiancia de salida en un punto y en la dirección ω es la misma que llega a otro punto x por la dirección contraria $-\omega$

$$L(x \leftarrow -\omega) = L(y \to \omega) \tag{1.3}$$

para cualquier par de puntos x e y mutuamente visibles, y siendo ω la dirección que parte de y y apunta a x. Esta propiedad de la radiancia es consecuencia de dos propiedades. La primera es que la radiancia se *conserva en línea recta*

$$L(x \leftarrow \omega) = L(y \leftarrow \omega)$$

La segunda es que la radiancia de entrada es la misma que la de salida (Shirley [Shi91], págs. 32–33)

$$L(x \to \omega) = L(x \leftarrow -\omega)$$

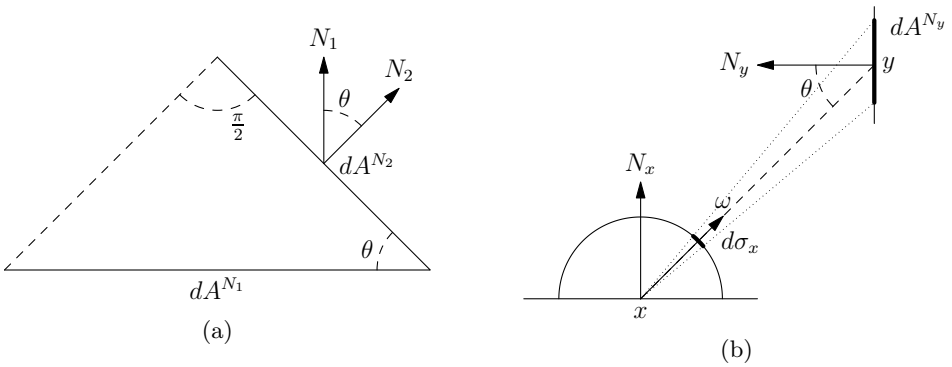


Figura 1.3: Fig. (a). Proyección del plano de dA^{N_2} sobre el plano de dA^{N_1} en la dirección N_2. Fig. (b). Ángulo sólido subtendido por la superficie dA^{N_y}.

Sin embargo, la relación 1.3 se cumple en todo punto del espacio excepto en puntos sobre la superficie de los objetos (este caso se tratará en la sección 1.4). Además, es necesario que no exista ningún tipo de materia en el espacio que hay entre objetos, ya sea, por ejemplo, humo, polvo o niebla[7]. Por tanto, asumiremos que entre las superficies de los objetos solo existe el vacío.

1.3.2. Derivación de la Irradiancia

A partir de la radiancia se obtienen el resto de magnitudes de la luz. En esta sección vamos a ver cómo se obtiene la irradiancia sobre los puntos de la superficie de los objetos de la escena a partir de la radiancia, aunque antes tenemos que establecer las siguientes relaciones entre medidas:

- Relación entre el área de una superficie y su proyección sobre otra. Supongamos que colocamos dos superficies con diferente orientación sobre un mismo punto x. Si proyectamos ortogonalmente una superficie sobre otra (figura 1.3a), la relación entre sus áreas es el coseno del ángulo entre sus vectores normales

$$\frac{dA^{N_2}(x)}{dA^{N_1}(x)} = |N_1 \cdot N_2| = |\cos(\theta)|$$

  El coseno del menor ángulo θ entre N_1 y N_2 se obtiene calculando el producto escalar de los dos vectores. El valor absoluto destaca el hecho de que no es importante la orientación de los vectores normales de las superficies, es decir, que la relación no varía si se multiplica alguno o los dos vectores por -1.

- Ángulo sólido subtendido por una superficie. La relación entre el área de una superficie en el punto y y el ángulo sólido subtendido por ella en un punto x (figura 1.3b), es la siguiente

$$\frac{d\sigma_x(\omega)}{dA^{N_y}(y)} = \frac{|N_y \cdot \omega|}{||x-y||^2} = \frac{|\cos(\theta)|}{||x-y||^2}$$

  donde $\omega = \frac{y-x}{||x-y||}$ es el vector unitario que apunta a y desde x. En este caso, a diferencia del anterior, el coseno del ángulo entre N_y y ω tiene que dividirse por la distancia entre los dos puntos al cuadrado.

[7]Estos materiales reciben el nombre de *medios participativos*.

Irradiancia direccional L'**,**[8] es la radiancia que llega a un punto x de una superficie de un objeto. Se define como la radiancia que llega al punto x multiplicado por el coseno del ángulo que forma la normal N_x en ese punto y la dirección ω

$$L'(x \leftarrow \omega) = \frac{d\Phi}{dA^{N_x}(x)\, d\sigma_x(\omega)} = |N_x \cdot \omega|\, L(x \leftarrow \omega)$$

Al igual que otras magnitudes, también podemos hablar de la radiancia que sale de esa superficie, denotándola como $L'(x \rightarrow \omega)$.

Irradiancia punto a punto L'**,**[9] es el flujo de energía que llega a un punto x de una superficie real desde otro punto y

$$L'(x \leftarrow y) = \frac{d\Phi}{dA^{N_x}(x)\, dA^{N_y}(y)}$$

Su relación con la radiancia viene expresada por

$$L'(x \leftarrow y) = G(x,y)\, L(x \leftarrow \omega) = G(x,y)\, L(y \rightarrow -\omega)$$

donde $\omega = \frac{y-x}{||y-x||}$ es el vector unitario que apunta a y desde x. Al término $G(x,y)$ se le conoce como *término geométrico* y depende de los cosenos de los ángulos que forman las normales con ω y del cuadrado de la distancia entre los puntos x e y

$$G(x,y) = \frac{|N_x \cdot \omega|\ |N_y \cdot \omega|}{||y-x||^2}\, V(x,y) = \frac{|N_x \cdot (y-x)|\ |N_y \cdot (y-x)|}{||y-x||^4}\, V(x,y)$$

Al término $V(x,y)$ se le llama *término de visibilidad* e indica si los dos puntos son mutuamente visibles, es decir, esta función vale 1 cuando los puntos son visibles y 0 cuando existe un objeto entre ellos.

Irradiancia total E**,**[10] es el flujo de energía total que llega a un punto de una superficie.

$$E(x) = \frac{d\Phi}{dA^{N_x}(x)} = \int_{\omega \in \mathcal{H}_x} L(x \leftarrow \omega)\, |N_x \cdot \omega|\, d\sigma_x(\omega)$$

donde $\mathcal{H}_x$ es una semiesfera, o hemisferio, situado en el punto x y cuyo polo se encuentra apuntado por la normal N_x en el punto x. Este conjunto $\mathcal{H}_x$ indica, en realidad, el conjunto de las direcciones por las que puede llegar radiancia al punto x, es decir, las direcciones que apuntan al exterior del objeto. En caso de que la energía sea saliente, esta magnitud se llama *radiosity*.

1.4. BRDF

La BRDF, del inglés *bidirectional reflectance distribution function*, (Nicodemus [Nic65]) es una función que relaciona la radiancia de salida con respecto a la irradiancia direccional de entrada

[8] En general, en los libros de Informática Gráfica se define la magnitud *irradiancia* como el flujo de energía por unidad de superficie (que nosotros hemos denominado *irradiancia total*). En esta tesis, vamos a llamar irradiancia a cualquier densidad de flujo de energía que llega a la superficie de un objeto. En concreto, definimos tres magnitudes: la irradiancia direccional, la irradiancia punto a punto y la irradiancia total.

[9] Aunque la irradiancia punto a punto y la direccional tienen la misma notación —la letra L'— designan cosas diferentes. Se usarán sus parámetros para distinguirlas. Así, la irradiancia direccional, $L'(x \leftarrow \omega)$, tiene como parámetros un punto x y una dirección ω, mientras que la irradiancia punto a punto, $L'(x \leftarrow y)$, tiene como parámetros dos puntos, x e y.

[10] Como ya se ha comentado, otros autores llaman a esta magnitud simplemente *irradiancia*.

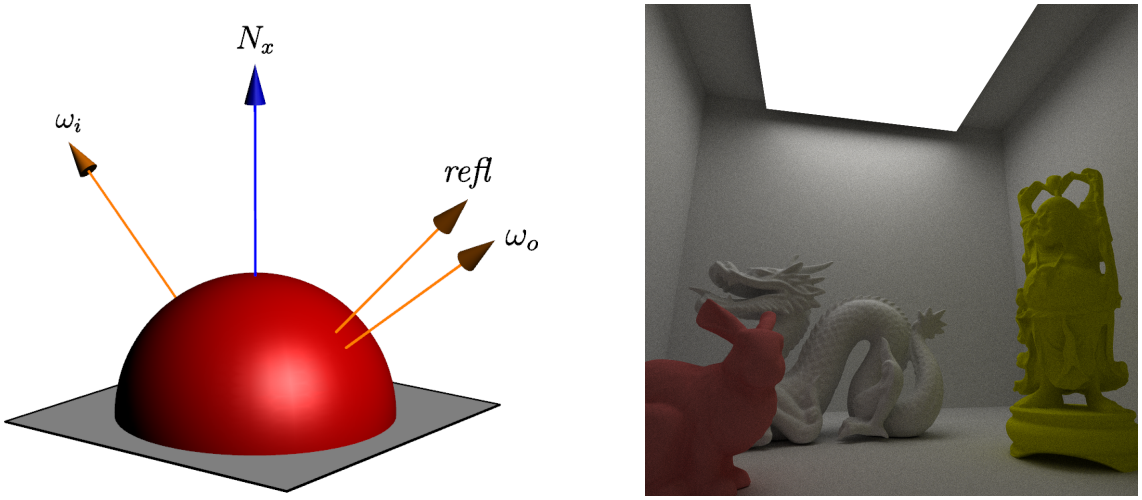


Figura 1.4: BRDF diffuse. En la imagen de la izquierda, la semiesfera roja muestra la radiancia de salida como consecuencia de la radiancia de entrada por ω_i. Como se puede apreciar, la radiancia de salida es independiente de la dirección de salida, ω_o. En la imagen de la derecha, se muestra una escena con tres figuras cuyos materiales son todos diffuse.

en puntos de la superficie de los objetos. Dicho con otras palabras, especifica completamente el proceso de difusión de la luz entre los objetos. Se define como la función f_r tal que

$$L(x \to \omega_o) = \int_{\omega_i \in \mathcal{H}_x} f_r(\omega_o, x, \omega_i) L'(x \leftarrow \omega_i)\, d\sigma_x(\omega_i)$$

Las BRDFs son parte de la información de la escena y son la manera de especificar las propiedades físicas de los materiales. Diferentes BRDFs dan diferentes apariencias a los objetos, simulando distintas propiedades físicas reales. Para que esta función sea físicamente correcta se deben cumplir las siguientes dos condiciones. La primera es que se debe satisfacer la *conservación de la energía*, es decir, la energía total saliente debe ser igual o menor que la energía entrante. La siguiente propiedad es condición suficiente para que se conserve la energía

$$\int_{\omega_o \in \mathcal{H}_x} f_r(\omega_o, x, \omega_i) |N_x \cdot \omega_o|\, d\sigma_x(\omega_o) \leq 1 \tag{1.4}$$

La segunda condición consiste en que la BRDF sea *simétrica*, es decir, que se puedan intercambiar las direcciones de entrada y de salida

$$f_r(\omega_o, x, \omega_i) = f_r(\omega_i, x, \omega_o)$$

1.4.1. BRDF Diffuse

Una superficie *diffuse*, también llamada *lambertiana*, es aquella en la que su apariencia no depende del ángulo de visión (figura 1.4). Esta BRDF imita superficies que son mate y sin brillo. La forma de simular esto es estableciendo la BRDF como una constante

$$f_r(\omega_o, x, \omega_i) = k$$

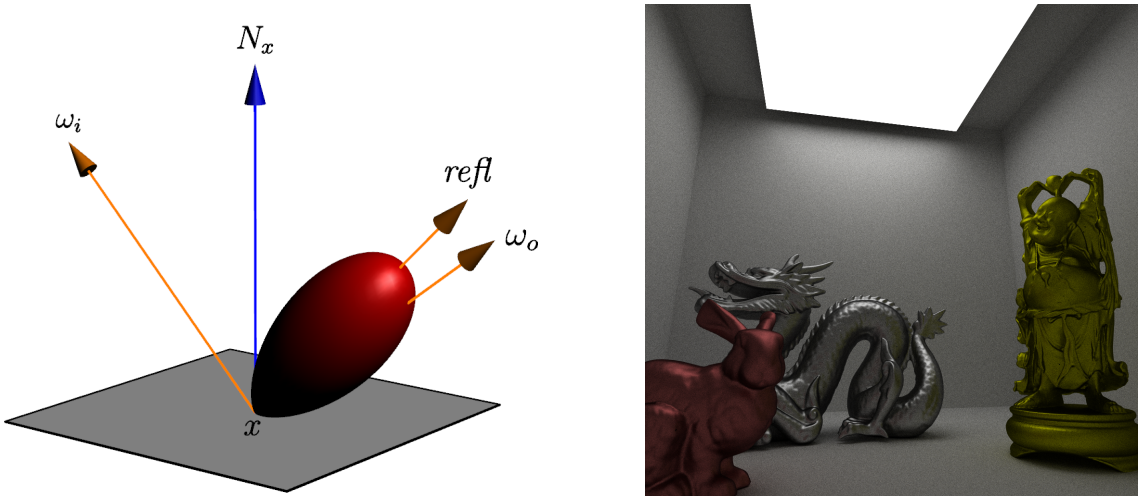


Figura 1.5: BRDF glossy. En la imagen de la izquierda, el lóbulo rojo muestra la radiancia de salida como consecuencia de la radiancia de entrada por ω_i. Como se puede apreciar, la radiancia de salida es más intensa en direcciones cercanas a la reflexión perfecta, *refl*. En la imagen de la derecha, se muestra una escena con tres figuras cuyos materiales son todos glossy.

La constante k no puede ser cualquier valor, sino solo aquellos que cumplan la conservación de la energía (ecuación 1.4)

$$\int_{\omega_o \in \mathcal{H}_x} k \left| N_x \cdot \omega_o \right| \, d\sigma_x(\omega_o) = k \int_0^{\pi/2} \int_0^{2\pi} \cos(\theta) \sin(\theta) \, d\phi \, d\theta = k\pi \leq 1$$

Tomándose $k = k_d/\pi$, con $k_d \in [0, 1]$, se satisface la ecuación 1.4, quedando la BRDF diffuse como

$$f_r(\omega_o, x, \omega_i) = \frac{k_d}{\pi}$$

1.4.2. BRDF Glossy

Un objeto *glossy* es aquel que refleja la luz en un rango de direcciones alrededor de la dirección de reflexión perfecta (figura 1.5). Un objeto con estas propiedades parece una superficie pulida, si el rango de direcciones es muy cerrado, o metálico, si el rango de direcciones es más abierto.

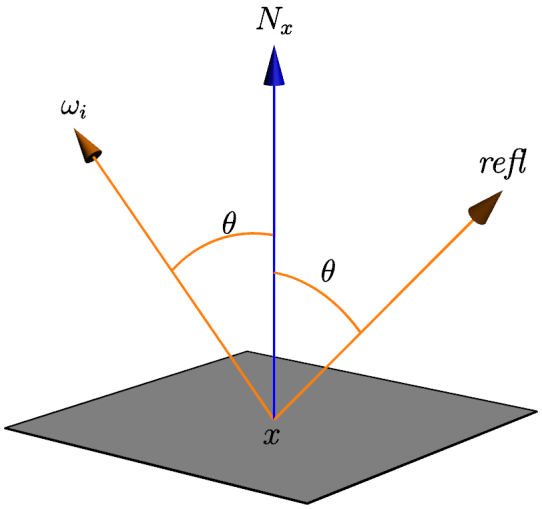
Definimos la BRDF glossy como

$$f_r(\omega_o, x, \omega_i) = \begin{cases} k \left(refl(x, \omega_i) \cdot \omega_o \right)^\alpha & \text{si } \left(refl(x, \omega_i) \cdot \omega_o \right) > 0 \\ 0 & \text{si } \left(refl(x, \omega_i) \cdot \omega_o \right) \leq 0 \end{cases}$$

donde α es un real positivo que controla la anchura del lóbulo de direcciones, k es una constante, y $refl(x, \omega_i)$ es la dirección de reflexión perfecta de ω_i en el punto x, definida como

$$refl(x, \omega_i) = 2(N_x \cdot \omega_i)N_x - \omega_i$$

Ya que la propiedad de la conservación de la energía se debe cumplir para toda entrada ω_i, es suficiente determinar las condiciones que debe cumplir k cuando el vector *refl* coincide con la normal N_x, es decir, cuando la radiancia de entrada es perpendicular a la superficie ($refl(x, \omega_i) = \omega_i = N_x$)



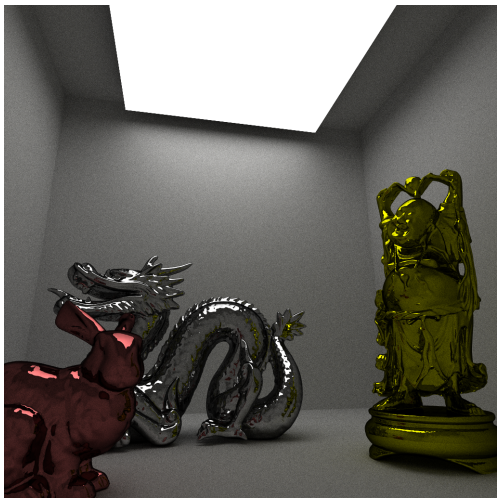


Figura 1.6: BRDF specular. En la imagen de la izquierda, toda la radiancia de entrada por ω_i sale por la dirección de reflexión perfecta, *refl*. En la imagen de la derecha, se muestra una escena con tres figuras cuyos materiales son todos specular.

$$\int_{\omega_o \in \mathcal{H}_x} f_r(\omega_o, x, \omega_i) |N_x \cdot \omega_o| \, d\sigma_x(\omega_o) = \int_0^{2\pi} \int_0^{\pi/2} k \cos^{\alpha+1}(\theta) \sin(\theta) \, d\theta \, d\phi = k \frac{2\pi}{\alpha + 2} \leq 1$$

Tomando $k = k_g \frac{\alpha+2}{2\pi}$, con $k_g \in [0, 1]$, se cumple la ecuación 1.4, quedando la BRDF glossy como

$$f_r(\omega_o, x, \omega_i) = \begin{cases} k_g \frac{\alpha+2}{2\pi} (refl(x, \omega_i) \cdot \omega_o)^\alpha & \text{si } (refl(x, \omega_i) \cdot \omega_o) > 0 \\ 0 & \text{si } (refl(x, \omega_i) \cdot \omega_o) \leq 0 \end{cases}$$

1.4.3. BRDF Specular

La idea detrás de la BRDF *specular* es simular un espejo perfecto. En este caso, toda la radiancia de entrada en un punto sale reflejada por el vector de reflexión perfecto (figura 1.6), por tanto, se podría definir como

$$f_r(\omega_o, x, \omega_i) = \begin{cases} k & \text{si } \omega_o = refl(x, \omega_i) \\ 0 & \text{en otro caso} \end{cases}$$

Sin embargo, la integral de esta función sobre el hemisferio es cero, lo que implicaría que la radiancia de salida sería también de cero. Este comportamiento no es el que queremos simular. Una posible solución solo para esta BRDF sería prescindir de la integral cuando se define la radiancia de salida. Sin embargo, para evitar tratar este caso como una excepción entre las BRDFs, se suele introducir la función *delta de Dirac* δ. La delta de Dirac se define como una función de medida tal que

$$\delta_a(A) = \begin{cases} 1 & \text{si } a \in A \\ 0 & \text{en otro caso} \end{cases}$$

donde A es un conjunto cualquiera, aunque es habitual definirla de manera informal como

$$\delta_a(x) = \begin{cases} \infty & \text{si } a = x \\ 0 & \text{en otro caso} \end{cases}$$

Esta delta tiene la siguiente propiedad

$$\int f(x)\delta_a(x)\ d\mu = f(a)$$

lo que permite integrar funciones que son nulas en todo el dominio excepto en un conjunto finito de valores (ver [CK04], págs. 68-69 y 106).

Usando la delta de Dirac, se define la BRDF especular como

$$f_r(\omega_o, x, \omega_i) = k\,\delta_{refl(x,\omega_i)}(\omega_o)$$

Para que se cumpla la conservación de la energía (ecuación 1.4)

$$\begin{aligned}&\int_{\omega_o\in\mathcal{H}_x} f_r(\omega_o, x, \omega_i)|N_x\cdot\omega_o|\ d\sigma_x(\omega_o)\\ &= k\int_0^{2\pi}\int_0^{\pi/2}\delta_{\theta_{refl}}(\theta)\delta_{\phi_{refl}}(\phi)\cos(\theta)\sin(\theta)\ d\theta\,d\phi\\ &= k\cos(\theta_{refl})\sin(\theta_{refl}) \le 1\end{aligned}$$

donde θ_{refl} y ϕ_{refl} son las coordenadas esféricas del vector de reflexión perfecta de ω_i. Tomando $k = \frac{k_s}{\cos(\theta_{refl})\sin(\theta_{refl})}$, con $k_s \in [0,1]$, la BRDF especular queda como

$$f_r(\omega_o, x, \omega_i) = \frac{k_s}{\cos(\theta_{refl})\sin(\theta_{refl})}\,\delta_{refl(x,\omega_i)}(\omega_o)$$

1.5. El Problema del Transporte de Luz

El objetivo de la Informática Gráfica es obtener imágenes realistas basadas en el modelo de partículas anteriormente descrito. Para ello, tenemos que relacionar la luz que es emitida por las luminarias con la que llega a la cámara. La BRDF pone en relación la radiancia de entrada y de salida solo en puntos de la superficie de los objetos. La radiancia saliente, debida solo a la difusión, será, por tanto, la integral de toda la irradiancia direccional entrante multiplicada por la BRDF en dicho punto

$$L(x\to\omega_o) = \int_{\omega_i\in\mathcal{H}_x} f_r(\omega_o, x, \omega_i)\,L(x\leftarrow\omega_i)\,|N_x\cdot\omega_i|\ d\sigma_x(\omega_i) \tag{1.5}$$

La ecuación 1.5 solo tiene en cuenta la radiancia entrante y no la que pueda ser generada sobre la superficie del propio objeto, en caso de que este sea una luminaria. Por tanto, la radiancia emisora, $L_e(x\to\omega_o)$, tiene que ser sumada a la radiancia de reflexión para obtener la radiancia total de salida. Al igual que los objetos de la escena y sus materiales, la función L_e se asume que viene definida como parte de la escena, ya que indica la cantidad de luz que emiten los objetos.

Por otro lado, buscamos poner en relación la misma función en la ecuación 1.5 para obtener una ecuación recursiva, es decir, queremos relacionar solo radiancia saliente con radiancia saliente. Para ello, usamos la propiedad de que la radiancia se conserva en una línea recta (ecuación 1.3) para relacionar ambas funciones. Con todo esto, derivamos la *ecuación de renderizado en forma direccional*:

$$L(x\to\omega_o) = L_e(x\to\omega_o) + \int_{\omega_i\in\mathcal{H}_x} f_r(\omega_o, x, \omega_i)\,L(z\to-\omega_i)\,|N_x\cdot\omega_i|\ d\sigma_x(\omega_i) \tag{1.6}$$

donde $z = rc(x, \omega_i)$ es la función *ray casting*, que devuelve el punto de intersección más cercano desde el punto x en la dirección ω_i. El valor I_j del píxel j-ésimo de la cámara se obtiene integrando la irradiancia total que llega a la cámara multiplicada por su función de importancia $W_e^{(j)}$

$$I_j = \int_{x \in \mathcal{C}} W_e^{(j)}(x) E(x)\, dA^{N_x}(x) = \int_{x \in \mathcal{C}} \int_{\omega \in O_x} W_e^{(j)}(x) L(x \leftarrow \omega) |N_x \cdot \omega|\, d\sigma_x(\omega)\, dA^{N_x}(x) \quad (1.7)$$

donde $\mathcal{C}$ es el plano de la película y O_x son solo aquellas direcciones que comienzan en x y pasan por la apertura de la cámara. Por simplicidad, asumiremos que la película es parte de los objetos de la escena, es decir, $\mathcal{C} \subset \mathcal{M}$. La función de importancia $W_e^{(j)}$ indica la parte de la película que influye al píxel j. Esta función suele ser cero en todo el plano de la película excepto en una región alrededor del sensor.

Resolver las ecuaciones 1.6 y 1.7 implica resolver el problema del transporte de luz y, por tanto, la obtención de una imagen físicamente correcta. Sin embargo, la ecuación de renderizado en su forma direccional (ecuación 1.6) solo proporciona una visión local y parcial de la contribución de las luminarias a la imagen final. Recordemos que las partículas se generan en las fuentes de luz y rebotan por la escena hasta que son absorbidas o detectadas por la cámara. Por tanto, para conocer la contribución completa de un haz de luz a un píxel es necesario desplegar la ecuación recursiva 1.6 hasta llegar al caso base, es decir, aquel en el que la radiancia de salida coincide con la radiancia de emisión.

Una forma alternativa de ver el transporte de luz es considerar cada haz de luz como un todo que comienza en una luz y termina en la cámara. Definimos una *ruta* (*path* en inglés) de longitud n, como una sucesión finita de n puntos $x_0 \dots x_n$ sobre la superficie de los objetos de la escena, $x_i \in \mathcal{M}$. La *contribución* de una ruta es la irradiancia punto a punto que llega a su primer punto, x_0, solo desde el último, x_n. La expresión para un píxel es, por tanto, la integral sobre la contribución de todas las rutas que comienzan en una fuente de luz y terminan en la cámara. A esa ecuación la llamaremos *ecuación de renderizado en forma de rutas*.

Para derivar esta ecuación tenemos que derivar antes la *ecuación de renderizado en forma de tres puntos* (E. Veach [Vea98], págs. 220–222), que relaciona la irradiancia punto a punto que llega a un punto x desde otro y a partir de un tercero z

$$L'(x \leftarrow y) = G(x, y) \int_{z \in \mathcal{M}} f_r(\omega_o, y, \omega_i)\, L'(y \leftarrow z)\, G(y, z)\, dA^{N_z}(z) \quad (1.8)$$

donde los vectores ω_i y ω_o son los vectores de entrada y de salida, obtenidos a partir de los puntos x, y y z (figura 1.7a).

A partir de la ecuación 1.8 podemos obtener la irradiancia total de un conjunto de rutas de una longitud fija n, desplegando la ecuación n veces. Por ejemplo (figura 1.7b), la radiancia que llega a un punto x_0 desde todas las rutas de longitud 3 ($x_0 x_1 x_2 x_3$) es

$$E(x_0) = \int_{x_1 \in \mathcal{M}} L'(x_0 \leftarrow x_1)\, dA^{N_1}(x_1) = \int_{x_1 \in \mathcal{M}} G(x_0, x_1) \int_{x_2 \in \mathcal{M}} f_r(\omega_1, x_1, -\omega_2) G(x_1, x_2) \int_{x_3 \in \mathcal{M}} f_r(\omega_2, x_2, -\omega_3) G(x_2, x_3) L_e(x_3 \to \omega_3)\, dA^{N_3}(x_3)\, dA^{N_2}(x_2)\, dA^{N_1}(x_1) \quad (1.9)$$

En general, la contribución f de una ruta de longitud n, obtenida a partir de la ecuación 1.9, se define como

$$f(x_0 \dots x_n) = \left(\prod_{i=1}^{n-1} G(x_{i-1}, x_i) f_r(\omega_i, x_i, -\omega_{i+1}) \right) G(x_{n-1}, x_n)\, L_e(x_n \to \omega_n) \quad (1.10)$$

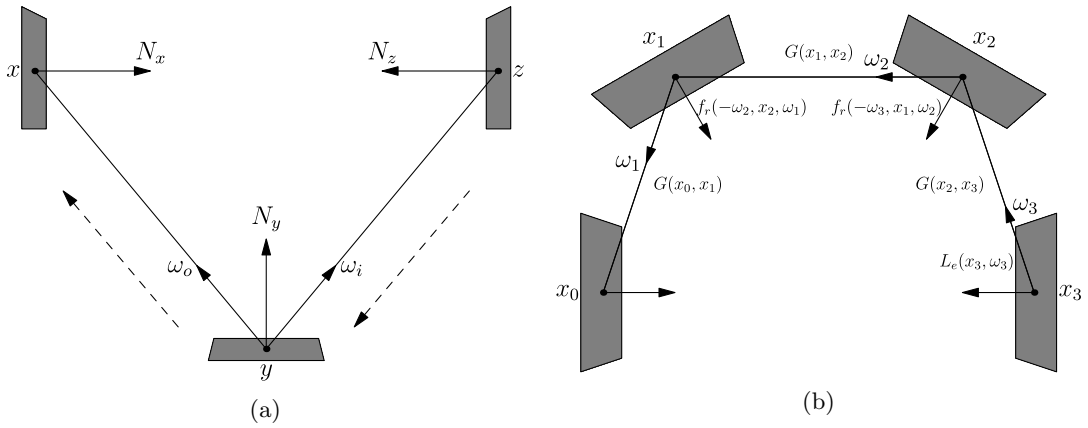


Figura 1.7: Fig. (a). Intercambio de radiancia entre el punto z y el x a través de otro punto y. Fig. (b). Ruta de longitud 3. La contribución de la ruta $x_0x_1x_2x_3$ es el producto de todos los factores L_e, G y f_r que aparecen en el dibujo.

donde cada ω_i es un vector unitario que comienza en x_i y termina en x_{i-1}.

Una imagen se forma a partir de toda la energía que llega a la película de la cámara. Llamemos Ω_n al conjunto de todas las rutas de longitud n que llegan a la cámara desde alguna fuente de luz, es decir, todas las rutas cuyo primer punto x_0 se encuentra en el plano de la cámara, el segundo x_1 es un punto visible desde la cámara, el último x_n se encuentra en una luz, y el resto son puntos sobre la superficie de los objetos de la escena. Al conjunto de todas las rutas lo definimos como

$$\Omega = \bigcup_{n\geq 1} \Omega_n \quad \text{donde} \quad \Omega_n = \overbrace{\mathcal{M} \times \cdots \times \mathcal{M}}^{n+1 \text{ veces}}$$

La *ecuación de renderizado en forma de rutas* queda, por tanto, así

$$I_j = \int_{\pi\in\Omega} W_e^{(j)}(\pi)\, f(\pi)\, d\mu(\pi) \tag{1.11}$$

donde

$$d\mu(x_0 \ldots x_n) = dA(x_0) \ldots dA(x_n)$$

y f es la contribución de la ruta (ecuación 1.10). La ecuación 1.11 representa, de manera sucinta, la contribución de todas las fuentes de luz al píxel j-ésimo de la cámara, según el modelo de partículas de la sección 1.2. Cualquier método empleado para resolver o aproximar la ecuación anterior para cada píxel da como resultado una imagen en forma de matriz bidimensional de colores. Ya que las partículas simulan el comportamiento de la luz en el mundo real, las imágenes obtenidas tienen la apariencia realista de una foto.

Sin embargo, las integrales de la ecuación 1.11 son difíciles de resolver analíticamente, por lo que suelen aproximarse con métodos numéricos. La manera más habitual de hacerlo es usando el Método de Monte Carlo, basado en la evaluación de rutas puntales generadas aleatoriamente. Antes de nada, tenemos que proveernos de algunos recursos matemáticos (sección 1.6) para poder entender esta aproximación (sección 1.7).

1.6. Método de Monte Carlo

Los *métodos estadísticos* son aquellos métodos numéricos que aproximan la solución de un problema mediante la evaluación de muestras aleatorias. Estos métodos eran ya conocidos desde hace mucho tiempo como, por ejemplo, el método de las *agujas de Buffon*[11], usado para aproximar el valor de π. El procedimiento consiste en lanzar agujas de longitud L sobre un suelo en el que están dibujadas líneas paralelas separadas entre sí una distancia $D > L$. Repitiendo el experimento muchas veces y contando el número de agujas que han cruzado alguna línea (n_c) respecto del total lanzadas (n_d) se obtiene una aproximación de π como

$$\pi \approx \frac{2L}{(n_c/n_d)D}$$

Un inconveniente de estos métodos es que son necesarias muchas muestras para obtener una buena aproximación del valor deseado. Al tener que ser realizados estos cálculos a mano, cayeron en desuso hasta que aparecieron los primeros ordenadores. Durante la Segunda Guerra Mundial y después, los métodos estadísticos, renombrados como *Métodos de Monte Carlo* (abreviado como *MC*) por N. Metrópolis[12], como él mismo afirma [Met87], fueron usados para simular el transporte de neutrones en las reacciones de fisión atómicas. Posteriormente, científicos de categoría como John von Neumann[13], Stanislaw Ulam[14], Enrico Fermi[15] y otros aplicaron los ordenadores de la época para resolver otros problemas de la física mediante MC, viendo su utilidad práctica. Para más información sobre la historia de los métodos de Monte Carlo se pueden consultar [Met87] y el capítulo 1 de [DS12].

1.6.1. Probabilidad

Sea Ω un conjunto cualquiera al que vamos a llamar *espacio muestral*. Un *evento* A es un subconjunto del espacio muestral Ω. Una función de probabilidad P es cualquier función de medida sobre Ω que cumpla los siguientes axiomas:

1. $P(\Omega) = 1$. El evento total es un evento seguro.
2. $P(A) \geq 0$, para todo evento $A \subset \Omega$. La probabilidad es siempre una medida positiva.
3. $P\left(\bigcup_i A_i\right) = \sum_i P(A_i)$, si todos los eventos son disjuntos dos a dos.

Si todo lo anterior se cumple, a la tupla (Ω, P) se le llama *espacio de probabilidad*[16]. La probabilidad es, por tanto, una función que asigna a cada evento una "certeza" sobre la ocurrencia del mismo. El valor de la probabilidad se encuentra entre cero (un *evento imposible*) y uno (un *evento seguro*).

Sea $X \in \Omega$ un elemento aleatorio del espacio muestral. Esta variable recibe el nombre de *variable aleatoria*. La probabilidad de que la variable aleatoria pertenezca a un cierto evento es la probabilidad del propio evento

$$P(X \in A) = P(A)$$

[11] Georges Louis Leclerc, Conde de Buffon, 1707–1788.
[12] Nicholas Constantine Metropolis, 1915–1999.
[13] John von Neumann, 1903–1957.
[14] Stanislaw Marcin Ulam, 1909–1984.
[15] Enrico Fermi, 1901–1954.
[16] Desde una perspectiva matemática (ver [CK04] para más detalles), es necesario que el conjunto de todos los eventos sea una σ-álgebra de conjuntos, es decir, un conjunto no vacío y cerrado respecto de la unión contable (finita o infinita numerable) y la complementación de conjuntos. Esto permite excluir del desarrollo posterior subconjuntos de Ω que no sean medibles por la probabilidad P.

Si el espacio muestral es la recta real y el evento es un intervalo $[a, b]$, se suele emplear la siguiente notación para referirse a la probabilidad de que un número real Y tomado aleatoriamente pertenezca al intervalo

$$P(a \leq Y \leq b) = P([a, b])$$

En ciertos casos, podemos preguntar sobre la probabilidad de un evento A pero restringiéndolo a que solo se encuentre dentro de otro evento B. Esta nueva probabilidad recibe el nombre de *probabilidad condicionada de A dado B*, $P(A|B)$, y se define como

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

para cualquier par de eventos A y B, siempre y cuando $P(B) \neq 0$. De la definición anterior se deduce la siguiente relación entre $P(A|B)$ y $P(B|A)$,

$$P(A|B)\, P(B) = P(B|A)\, P(A)$$

1.6.2. Función de Densidad de Probabilidad

Sea (Ω, P) un espacio de probabilidad y sea μ otra función de medida sobre Ω. Siempre que P y μ sean absolutamente continuas ($P \ll \mu$), existe una función de densidad p entre la probabilidad P y la medida μ

$$p = \frac{dP}{d\mu}$$

La función p se llama *función de densidad de probabilidad* o *pdf* (de *probability density function*). La pdf p tiene las siguientes propiedades

1. $p(x) \geq 0$, para todo $x \in \Omega$. La pdf es siempre positiva.
2. $\displaystyle\int_{x \in \Omega} p(x)\, d\mu(x) = 1$. El evento total es siempre un evento seguro.

Es posible que la probabilidad P no se defina directamente, sino a través de la medida μ y la pdf p. En ese caso, la probabilidad de un evento A se obtiene como

$$P(A) = \int_{x \in A} p(x)\, d\mu(x)$$

Si el espacio muestral es la recta real $\mathbb{R}$, entonces se suele usar la longitud de intervalos l como medida

$$l([a, b]) = b - a$$

Así, la probabilidad de que un número real Y pertenezca a un intervalo es

$$P(Y \in [a, b]) = P(a \leq Y \leq b) = \int_{y \in [a,b]} p(y)\, dl(y) = \int_a^b p(y)\, dy$$

Si el conjunto Ω es el producto de dos espacios muestrales, $\Omega = \Omega_1 \times \Omega_2$, entonces podemos definir relaciones entre las pdfs de estos espacios de probabilidad. Se llama *función de densidad conjunta*, $p(x, y)$, a la función que cumple

$$P\left((X, Y) \in D\right) = P(D) = \int_{(x,y) \in D} p(x, y)\, d\mu_1(x)\, d\mu_2(y)$$

donde $D \subset \Omega$, y μ_1 y μ_2 son medidas de Ω_1 y Ω_2, respectivamente. La *función de densidad marginal* de Ω_1 se define como

$$p(x) = \int_{y \in \Omega_2} p(x, y)\, d\mu_2(y)$$

y de manera equivalente la función de densidad marginal para Ω_2. La *función de densidad condicional* relaciona la función de densidad marginal con la conjunta como

$$p(y|x) = \frac{p(x, y)}{p(x)}$$

Por tanto,

$$p(x, y) = p(y|x)\, p(x) = p(x|y)\, p(y)$$

1.6.3. Función de Distribución Acumulada

En el caso de que el espacio muestral sea la recta real, $\Omega = \mathbb{R}$, se suele definir la *función de distribución acumulada* o *cdf* (de *cumulative distribution function*) como

$$F(y) = P(Y \leq y)$$

Las propiedades de esta función son

1. Si $x \leq y$ entonces $F(x) \leq F(y)$. F es monótona no decreciente.
2. $\lim_{y \to -\infty} F(y) = 0$. El límite hacia $-\infty$ corresponde con la probabilidad del conjunto vacío.
3. $\lim_{y \to \infty} F(y) = 1$. El límite hacia ∞ corresponde con la probabilidad del espacio total.

La relación de la cdf con la pdf es

$$p(y) = \frac{dF(y)}{dy}$$

Por tanto, la probabilidad de un intervalo se expresa directamente con la cdf como

$$P([a, b]) = \int_a^b p(y)\, dy = F(b) - F(a)$$

1.6.4. Esperanza y Varianza

Sea f una función que asigna a cada elemento de un espacio muestral un valor real. Sea p la pdf de la variable aleatoria X, lo que denotaremos mediante $X \sim p$. Sobre la variable aleatoria real $Y = f(X)$ podemos definir dos operaciones: su *esperanza* y su *varianza*. La esperanza se denota con la letra E y se define como

$$E[Y] = E[f(X)] = \int_{x \in \Omega} f(x)\, p(x)\, d\mu(x)$$

La varianza de una variable aleatoria se define como la esperanza del error al cuadrado

$$V[Y] = E[(Y - E[Y])^2] = E[Y^2] - E[Y]^2$$

La raíz cuadrada de la varianza recibe el nombre de *desviación típica* y se suele usar como medida del *error esperado*

$$\sigma[Y] = \sqrt{V[Y]}$$

1.6.5. Relación entre Variables Aleatorias

Sea X una variable aleatoria real con pdf p_X y cdf F_X. Sea $Y = f(X)$ otra variable aleatoria real resultante de la aplicación de f a X. Si f es continua y creciente, entonces la probabilidad de que Y pertenezca a un intervalo se expresa en función de la probabilidad de X como

$$P(Y \in (-\infty, f(a)]) = P(X \in (-\infty, a])$$

para todo $a \in \mathbb{R}$. A partir de la propiedad anterior, es fácil demostrar que la cdf de ambas variables aleatorias es la misma

$$F_Y(y) = F_Y(f(x)) = P(Y \leq f(x)) = P(X \leq x) = F_X(x)$$

La pdf p_Y de Y se relaciona de manera igualmente fácil con p_X a partir de la derivada de F_Y y de la *regla de la cadena* (Pharr y Humphrey [PH10], págs. 660–661)

$$p_X(x) = \frac{dF_X(x)}{dx} = \frac{dF_Y(f(x))}{dx} = \frac{dF_Y(f(x))}{df}\,\frac{df(x)}{dx} = p_Y(f(x))\,\frac{df(x)}{dx}$$

$$\Rightarrow p_X(x) = p_Y(y)\,\frac{df(x)}{dx}$$

Supongamos ahora que tanto $X = (X_1, \ldots, X_n)$ como $Y = (Y_1, \ldots, Y_n)$ son vectores de variables aleatorias. Entonces la relación entre sus pdfs es

$$p_X(x) = p_Y(y)\left|\frac{\partial f}{\partial x_1 \ldots \partial x_n}\right|$$

donde $\left|\frac{\partial f}{\partial x_1 \ldots \partial x_n}\right|$ es el determinante de la matriz Jacobiana de f.

1.6.6. Método de Inversión

El objetivo del *método de inversión* consiste en obtener una variable aleatoria con una cdf deseada F_X, a partir de una *variable aleatoria uniforme*. Una variable aleatoria uniforme, denotada como ξ, es una variable aleatoria sobre $[0, 1]$ cuya pdf es constante ($p_\xi(x) = 1$) y su cdf es la identidad ($F_\xi(x) = x$). Además, suponemos que F_X cumple los mismos supuestos que los realizados sobre f en la sección 1.6.5, es decir, que sea continua y creciente.

Este método se basa en el hecho de que una cdf aplicada sobre la propia variable aleatoria resulta en una variable aleatoria uniforme. Como se puede ver, si $Y = F_X(X)$ y a es un valor real cualquiera que puede tomar X, entonces

$$F_Y(a) = P(Y \leq a) = P(X \leq F_X^{-1}(a)) = F_X(F_X^{-1}(a)) = a$$

y, por lo tanto, $Y = \xi$ es una variable aleatoria uniforme. Así, podemos obtener una variable aleatoria X con una cdf deseada F_X simplemente aplicando la inversa de su cdf a una variable aleatoria uniforme

$$\xi = F_X(X) \Rightarrow X = F_X^{-1}(\xi)$$

Para aplicar este método, es necesario contar con variables aleatorias ξ uniformemente distribuidas en el intervalo $[0, 1]$. Vamos a suponer que cualquier generador de números pseudo-aleatorios imita en la práctica el comportamiento de ξ. En concreto, a lo largo de esta tesis hemos usado el generador de Park y Miller [PM88].

1.6.7. Integración por Monte Carlo

Consideremos la siguiente integral

$$I = \int_{x \in \Omega} f(x)\, d\mu(x)$$

Buscamos obtener el valor aproximado de I mediante el método probabilístico de Monte Carlo. Decimos que una variable aleatoria $\hat{I}$ es un *estimador unbiased* de I si cumple que su esperanza es el valor que se desea aproximar

$$E[\hat{I}] = I$$

Si $X \sim p$ es una variable aleatoria que toma valores sobre el dominio de la integral entonces la siguiente relación

$$\hat{I} = \frac{f(X)}{p(X)}$$

es un estimador unbiased de I. Esto se demuestra aplicando directamente la definición de la esperanza

$$E[\hat{I}] = E\left[\frac{f(X)}{p(X)}\right] = \int_{x \in \Omega} \frac{f(x)}{p(x)}\, p(x)\, d\mu(x) = I$$

Se debe cumplir que $p(x) \neq 0$ cuando $f(x) \neq 0$, para todo x en el dominio de la integral. De esta forma se asegura que la variable aleatoria X no "deja fuera" valores del dominio que se requieren para la integración.

Sean $\hat{I}_1, \ldots, \hat{I}_N$ estimadores unbiased de I, es decir, $E[\hat{I}_i] = I$ para cada $\hat{I}_i$ y sea

$$\hat{F}_N = \frac{1}{N} \sum_{i=1}^{N} \hat{I}_i$$

la media aritmética de los N estimadores anteriores. Se cumple que la media aritmética de estos N estimadores es también un estimador unbiased de la integral I

$$E[\hat{F}_N] = E\left[\frac{1}{N} \sum_{i=1}^{N} \hat{I}_i\right] = \frac{1}{N} \sum_{i=1}^{N} E[\hat{I}_i] = I$$

Por consiguiente, el proceso de aproximar una integral por el Método de Monte Carlo es el siguiente. Primero, se eligen aleatoriamente N puntos dentro del dominio de la integral. El proceso de obtener estos puntos aleatorios se conoce como *muestreo*, y cada uno de ellos es una *muestra*. Posteriormente, se evalúa la función f en cada uno de esos puntos y se divide por su densidad de probabilidad. La media aritmética de todos ellos será una aproximación de la integral. Cuanto más grande sea el valor de N, el valor de $\hat{F}_N$ más se aproxima al de la integral I. El teorema conocido como *ley de los grandes números* viene a confirmar este hecho, que suele expresarse como

$$P\left(\lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} \hat{I}_i = I\right) = 1$$

Una forma de medir el error entre el estimador $\hat{I}$ y el valor I que aproxima es analizando la esperanza de la distancia entre ellos, es decir, $E[\hat{I} - I]$. Sin embargo, por ser $\hat{I}$ un estimador unbiased, este valor es siempre cero

$$E[\hat{I} - I] = E[\hat{I}] - E[I] = 0$$

La idea detrás de este hecho es que el estimador $\hat{I}$ se desvía, de media, la misma distancia de I tanto por exceso como por defecto. Por tanto, es más conveniente analizar la esperanza de un valor que sea siempre positivo. La manera habitual es usando la esperanza del cuadrado de la distancia, lo que corresponde con la varianza del estimador

$$E[(\hat{I} - I)^2] = V[\hat{I}] = E[\hat{I}^2] - E[\hat{I}]^2 = \int_{x \in \Omega} \frac{f^2(x)}{p(x)} \, d\mu - I^2$$

A la raíz cuadrada de la varianza, la desviación típica $\sigma[\hat{I}]$, se le considera el error de un estimador.

Si $\hat{F}_N$ es la media aritmética de N estimadores unbiased, todos con la misma varianza, entonces la varianza de la media disminuye un factor de N

$$V[\hat{F}_N] = V\left[\frac{1}{N}\sum_{i=1}^{N} \hat{I}_i\right] = \frac{1}{N} V[\hat{I}]$$

Por tanto, el error cuadrado medio disminuye en una tasa de $\sqrt{N}$

$$\sigma[\hat{F}_N] = \sigma\left[\frac{1}{N}\sum_{i=1}^{N} \hat{I}_i\right] = \sqrt{V\left[\frac{1}{N}\sum_{i=1}^{N} \hat{I}_i\right]} = \frac{1}{\sqrt{N}} \sigma[\hat{I}]$$

Cuanto más baja sea la varianza para un cierto estimador, se obtienen mejores aproximaciones de la integral con un número menor de muestras.

Una de las maneras de disminuir la varianza es usar variables aleatorias cuyas pdf tengan formas aproximadas a la función f. Esto se conoce como *important sampling*. Si la relación entre f y p es la constante k para todo x perteneciente al dominio de la integral

$$\frac{f(x)}{p(x)} = k$$

entonces la varianza del estimador $\hat{I} = \frac{f(x)}{p(x)}$ sería cero

$$V\left[\frac{f(X)}{p(X)}\right] = V\left[k\right] = 0$$

y una única muestra sería necesaria para obtener el valor de la integral I. Desafortunadamente, para que esto suceda, p tiene que ser igual a f/I, en donde se encuentra el valor desconocido I de la integral, lo que hace imposible aplicar este método directamente.

Sin embargo, se puede demostrar que, aunque f y p no sea proporcionales, cuanto más "parecidas" sean p y f, la varianza será más pequeña. Supongamos que la relación f/p está acotada por I más un cierto valor positivo ε. Entonces se puede demostrar que la varianza del estimador $\hat{I}$ es menor que $I\varepsilon$

$$\begin{array}{rcll} \frac{f(x)}{p(x)} & \leq & I + \varepsilon & \Rightarrow \\ \frac{f^2(x)}{p(x)} & \leq & (I+\varepsilon) f(x) & \Rightarrow \\ \int \frac{f^2(x)}{p(x)} d\mu & \leq & \int (I+\varepsilon) f(x) d\mu & \Rightarrow \\ \int \frac{f^2(x)}{p(x)} d\mu - I^2 & \leq & (I+\varepsilon) I - I^2 & \Rightarrow \\ V[\hat{I}] & \leq & I\varepsilon & \end{array}$$

Por tanto, a medida que ε se acerca a cero, las formas de las funciones f y p serán más parecidas y la varianza del estimador $\hat{I}$ será cada vez más pequeña.

Otra manera de entender el important sampling es pensar en un caso extremo en que p fuese muy diferente de f. Así, p sería grande donde f fuese pequeña, generándose muchas muestras en lugares donde el ratio f/p fuese muy pequeño. A medida que se generan más muestras aleatorias, aparecería alguna muestra con p pequeña pero f grande, obteniéndose una relación f/p muy grande. Al realizar la media aritmética de todas las muestras hasta ese momento se observa que se mantiene en un valor bajo pero, de vez en cuando, aumentaría mucho. Si N no es muy alto, el estimador será una mala aproximación de la integral como consecuencia de que la varianza es alta (se puede ver un ejemplo sencillo en P. Shirley [Shi94], pág. 11).

1.7. Aproximación de la Integral de Renderizado por Monte Carlo

Recordemos que en la sección 1.5 se ha establecido el problema de renderizar una imagen a partir de una escena tridimensional como una serie de integrales (una por píxel). Por tanto, el problema de calcular el color de cada píxel se ha convertido en el problema de resolver o aproximar cada una de las integrales de renderizado. Aunque la integral de renderizado de cada píxel se puede expresar de varias formas, usaremos preferentemente su forma de rutas (ecuación 1.11).

Resolver la integral de renderizado es difícil, por lo que se han desarrollado métodos numéricos para aproximar su valor. Una forma de aproximar esta integral es mediante el uso de los métodos conocidos como *cuadratura numérica*. Estos métodos se basan en dividir cada dimensión del dominio en intervalos y evaluar f para algún punto dentro de cada intervalo. Posteriormente, esos valores se combinan para obtener una aproximación de la integral. Sin embargo, los métodos de cuadratura numérica tienen el inconveniente conocido como la *maldición de la dimensión*: el número de muestras que se tienen que evaluar crece exponencialmente con el número de dimensiones. Por ejemplo, si el dominio tiene d dimensiones y cada una se divide en m secciones, el número de muestras total que se tiene que evaluar es de m^d, un número bastante grande para valores relativamente pequeños de m y d.

Los métodos probabilísticos de Monte Carlo tienen dos ventajas frente a los métodos de cuadratura numérica. Por un lado, la tasa de convergencia es siempre de $1/\sqrt{N}$ (sección 1.6.7), donde N es el número de muestras realizadas, independientemente de la dimensión del dominio. Por otro lado, el dominio de integración de la ecuación 1.11 tiene infinitas dimensiones, ya que no existe límite en la longitud de las rutas. Por estos motivos, Monte Carlo es el método preferido para la aproximación de las integrales de renderizado.

1.7.1. Muestreo de Rutas Aleatorias

Para resolver una integral por Monte Carlo es necesario obtener muestras aleatorias en el dominio de la integral. En el caso de la ecuación de renderizado (ecuación 1.11), su dominio es el conjunto de todas las rutas, por lo que necesitamos un procedimiento para generar rutas aleatorias. Una ruta se puede obtener muestreando secuencialmente sus puntos con respecto a la medida de área. El orden y la forma en que se muestreen los puntos va a determinar la pdf de la ruta completa. La pdf de una ruta es la densidad conjunta de sus puntos con respecto a la medida de área, y siempre es posible descomponer una pdf conjunta en un producto de densidades condicionales. Por ejemplo, para una ruta de longitud 3 ($\pi = x_0x_1x_2x_3$) una manera de descomponer su pdf es

$$p(x_0x_1x_2x_3) = p(x_0)\,\frac{p(x_0x_1)}{p(x_0)}\,\frac{p(x_0x_1x_2)}{p(x_0x_1)}\,\frac{p(x_0x_1x_2x_3)}{p(x_0x_1x_2)} = p(x_0)\,p(x_1|x_0)\,p(x_2|x_0x_1)\,p(x_3|x_0x_1x_2)$$

En la descomposición anterior, se puede ver que el primer punto se muestrea con su pdf, mientras que los siguientes se obtienen a partir de los anteriores. Esta descomposición no es única y otras

posibilidades surgen si se comienza desde otro punto. Por ejemplo, comenzando desde x_2 obtenemos

$$p(x_0x_1x_2x_3) = p(x_2)\,p(x_3|x_2)\,p(x_1|x_2x_3)\,p(x_0|x_1x_2x_3)$$

Una manera sencilla de generar rutas sería muestrear cada punto independientemente de los demás. De esta manera, cada densidad condicionada $p(x_i|x_{j_1}\dots x_{j_n}) = p(x_i)$, ya que cada punto x_i es independiente del resto $x_{j_1}\dots x_{j_n}$. Consecuentemente, la función de densidad de la ruta es ahora

$$p(x_0x_1x_2x_3) = p(x_0)\,p(x_1)\,p(x_2)\,p(x_3)$$

Sin embargo, esto tiene el inconveniente de que la mayoría de las rutas generadas tendrán contribución nula, como se puede observar por dos razones. Primero, solo las rutas que comienzan en la cámara y terminan en una luz contribuyen a la imagen. Segundo, aunque el primer punto se elija expresamente en el plano de la cámara, es necesario comprobar la visibilidad entre todas las parejas de puntos de la ruta para poder establecer los términos de visibilidad V. En caso de que no exista visibilidad entre dos puntos, su término V será 0 y toda la ruta tendrá contribución nula. Consecuentemente, es importante intentar generar solo las rutas que no tienen contribución nula, lo que se puede entender como una forma de important sampling, es decir, asignar pdf nula a rutas con contribución nula.

La forma habitual de muestrear rutas consiste en elegir un punto de la ruta previamente generado x y muestrear su hemisferio $\mathcal{H}_x$ para obtener el siguiente punto y. Sobre ese hemisferio $\mathcal{H}_x$ se elige una dirección ω aleatoria con una pdf $p_\sigma(\omega)$. Esa dirección se usa para encontrar el punto de intersección más cercano que será un nuevo punto $y = rc(x,\omega)$ de la ruta. La pdf del nuevo punto y se calcula a partir de la pdf con respecto al ángulo sólido de ω como

$$p(y|x) = \frac{dP}{d\sigma_x(\omega)} \cdot \frac{d\sigma_x(\omega)}{dA^{N_y}(y)} = p_\sigma(\omega) \cdot \frac{|N_y \cdot \omega|}{||x-y||^2}$$

Por tanto, la construcción de una ruta aleatoria se ha convertido en un proceso de dos fases. Supongamos que ya hemos obtenido un punto x de una ruta. En la primera fase, se elige una dirección aleatoria ω sobre el hemisferio $\mathcal{H}_x$ del punto x. En la segunda fase, hay que encontrar el punto de intersección con la escena del rayo que tiene por origen el punto x y dirección ω (función ray casting). Ese punto de intersección será un nuevo punto de la ruta que se está construyendo. Este proceso continuará de esta manera hasta que se haya formado una ruta entre la cámara y una fuente de luz, o la ruta termine, ya sea por un procedimiento *ad hoc* o por ruleta rusa (sección 1.7.4).

1.7.2. Muestreo del Hemisferio

Usando el método de inversión (sección 1.6.6) vamos a obtener muestras aleatorias sobre un hemisferio con una pdf dada. Consideremos la transformación usual de coordenadas esféricas a coordenadas cartesianas

$$T(\theta,\phi) = \begin{pmatrix} \sin(\theta)\cos(\phi) \\ \sin(\theta)\sin(\phi) \\ \cos(\theta) \end{pmatrix} \tag{1.12}$$

Esta transformación genera todos los puntos del hemisferio cuando los parámetros θ y ϕ recorren, respectivamente, los intervalos $[0,\pi/2]$ y $[0,2\pi]$ (figura 1.8).

Sea (Θ,Φ) un vector de dos variables aleatorias con pdf conjunta $p_{\Theta,\Phi}$. Si aplicamos la función T anterior a este par de variables aleatorias obtenemos puntos aleatorios en el hemisferio. La pdf de los puntos del hemisferio p_ω se relaciona con $p_{\Theta,\Phi}$ (sección 1.6.5) como

$$p_{\Theta,\Phi}(\theta,\phi) = p_\omega(T(\theta,\phi))\sin(\theta)$$

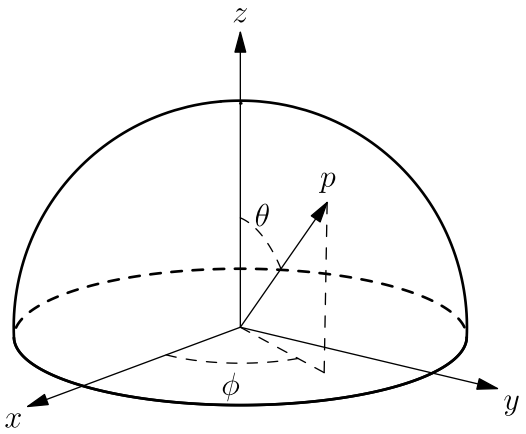


Figura 1.8: Coordenadas esféricas (θ, ϕ) del punto p en el hemisferio $\mathcal{H}_x$.

Para obtener muestras sobre el hemisferio con una p_ω deseada, tenemos primero que obtener las pdfs marginales y las cdfs de cada variable aleatoria Θ y Φ

$$p_\Theta(\theta) = \int_0^{2\pi} p_{\Theta,\Phi}(\theta,\phi)\, d\phi = \int_0^{2\pi} p_\omega(T(\theta,\phi)) \sin(\theta)\, d\phi$$
$$F_\Theta(\theta) = \int_0^{\theta} p_\Theta(\theta')\, d\theta'$$
$$p_\Phi(\phi) = \int_0^{\pi/2} p_{\Theta,\Phi}(\theta,\phi)\, d\theta = \int_0^{\pi/2} p_\omega(T(\theta,\phi)) \sin(\theta)\, d\theta$$
$$F_\Phi(\phi) = \int_0^{\phi} p_\Phi(\phi')\, d\phi'$$

Dadas dos variables aleatorias (ξ_1, ξ_2) uniformes, obtenemos muestras en el hemisferio con la pdf p_ω deseada si establecemos $\Theta = F_\Theta^{-1}(\xi_1)$ y $\Phi = F_\Phi^{-1}(\xi_2)$.

Para muestrear una BRDF diffuse, se tendría que muestrear el hemisferio con probabilidad proporcional al producto de la BRDF por la irradiancia de entrada

$$p(\omega_i) \propto \frac{k_d}{\pi} L(x \leftarrow \omega_i) \left| N_x \cdot \omega_i \right|$$

Sin embargo, eso no es posible ya que la radiancia de entrada L no es conocida. Por lo tanto, realizamos la suposición de que la radiancia de entrada es constante, así la probabilidad la hacemos proporcional al producto de la BRDF por el coseno. Aplicando el método de inversión obtenemos la dirección aleatoria en coordenadas esféricas (Θ, Φ) y en cartesianas (X, Y, Z).

$$\begin{array}{rclcrcl} \Theta &=& acos\sqrt{1-\xi_1} & & X &=& \sqrt{\xi_1}\cos(2\pi\xi_2) \\ \Phi &=& 2\pi\xi_2 & & Y &=& \sqrt{\xi_1}\sin(2\pi\xi_2) \\ & & & & Z &=& \sqrt{1-\xi_1} \end{array}$$

Para muestrear la BRDF glossy el procedimiento es semejante. Primero, se obtienen las variables aleatorias ponderadas con el lóbulo centrado en la normal, y posteriormente se rotan las muestras hasta que estén centradas en la dirección de reflexión de ω_o

$$\begin{array}{rclcrcl} \Theta &=& acos\ \sqrt[\alpha+1]{1-\xi_1} & & X &=& \sqrt[\alpha+1]{1-(1-\xi_1)^2}\cos(2\pi\xi_2) \\ \Phi &=& 2\pi\xi_2 & & Y &=& \sqrt[\alpha+1]{1-(1-\xi_1)^2}\sin(2\pi\xi_2) \\ & & & & Z &=& \sqrt[\alpha+1]{1-\xi_1} \end{array}$$

El muestreo de la BRDF Specular es sencillo debido a que la radiancia de entrada tiene solo una dirección de salida. Por tanto, el vector de salida ω_o corresponde al vector de reflexión perfecta del vector de entrada ω_i.

1.7.3. Métodos para la Generación de Rutas Aleatorias

Se han desarrollado diferentes algoritmos para la generación de rutas aleatorias. Básicamente, difieren en el orden en que se eligen los puntos aleatorios durante el muestreo de rutas. A continuación exponemos los más representativos.

Path Tracing

La forma de generar rutas que propone J. Kajiya [Kaj86] consiste en comenzar eligiendo el punto x_0 de la ruta directamente sobre la película de la cámara. Cada nuevo punto x_{i+1} de la ruta se obtiene trazando un rayo desde el último punto x_i ya generado, con la dirección obtenida muestreando la BRDF en dicho punto x_i. Las rutas obtenidas por este algoritmo, por tanto, recorren la escena en el orden inverso en el que lo hace la luz en nuestro modelo. Este algoritmo recibe el nombre de *Path Tracing* (PT).

Light Tracing

Dutre et al. [DLW93] generan rutas comenzando desde la luz, en un algoritmo llamado *Light Tracing*. Para aumentar la eficiencia del método se proponen dos optimizaciones. Primero, desde cada punto de las rutas se lanza un rayo hacia la cámara para comprobar su visibilidad. En caso afirmativo, se ha formado una ruta completa que contribuye a la imagen. En la segunda optimización, en vez de lanzar un único rayo hacia la cámara en los puntos de la ruta, se lanzan varios desde superficies cercanas a los puntos de las rutas.

Bidirectional Path Tracing

Lafortune y Willems [LW93] presentan una nueva forma de generar rutas completas. Así, se genera aleatoriamente una subruta desde la cámara y otra desde una fuente de luz. La unión de todos los puntos de una subruta con los de la otra forman una familia de rutas completas. Este método recibe el nombre de *Bidirectional Path Tracing* (BPT). Este algoritmo aumenta la probabilidad de formar rutas completas cuando la iluminación de la escena es, sobre todo, indirecta.

Veach y Guibas [VG94] también generan rutas comenzando desde la luz y desde la cámara. Además, proponen una forma de pesar la contribución de cada muestra teniendo en cuenta que cada ruta puede ser generada de varias maneras posibles.

Metropolis Light Transport

Veach y Guibas [VG97] aplican el muestreo del algoritmo de Metropolis-Hasting en la generación de rutas (*Metropolis Light Transport* o *MLT*). Una vez que una ruta con contribución no nula ha sido generada, se modifica para obtener otra. Esa nueva ruta es aceptada como otra muestra válida si su contribución a la imagen es mayor que la ruta de partida. Si su contribución es menor, puede ser aceptada o rechazada aleatoriamente.

En el caso límite, MLT genera muestras con una pdf proporcional a la contribución de todas las rutas a la imagen. Este tipo de muestreo es adecuado para escenas donde los otros métodos fallan al no obtener rutas con contribución no nula. Sin embargo, la pdf de las primeras muestras se ve muy influenciada por la muestra con que comienza el proceso, lo que se conoce como *start-up bias*. Cline et al. [CTE05] usan el muestreo de Metropolis-Hashing para distribuir la energía de

rutas previamente generadas por PT, en vez de usarlo directamente para generar rutas, evitando, de esa forma, el problema start-up bias.

1.7.4. Ruleta Rusa

Los algoritmos de generación de rutas producen rutas de cualquier longitud comenzando en la cámara y terminando en una fuente de luz. Sin embargo, es posible que algoritmos como PT o BPT necesiten muestrear muchos puntos de la escena hasta obtener una ruta completa. Por tanto, es deseable un procedimiento para terminar la generación de una ruta, liberando así la potencia de cómputo para generar otras.

Un método directo consiste en no considerar aquellas rutas que exceden una determinada longitud. Así, si durante el muestreo de los puntos de una ruta, su longitud sobrepasa un umbral y todavía no se ha completado, entonces se termina su generación y se comienza la generación de otra ruta nueva. Este método tiene el inconveniente de que restringe el dominio de la ecuación de renderizado, aproximando la integral solo para un conjunto menor de rutas. No obstante, esta técnica se puede usar cuando la iluminación de una escena es principalmente directa, por lo que las rutas largas contribuyen poco a la imagen final.

Arvo y Kirk [AK90] introducen en el campo de la Informática Gráfica el método de terminar rutas conocido como *ruleta rusa*, previamente desarrollado para la simulación del transporte de neutrones. Este método toma probabilísticamente la decisión de seguir extendiendo una ruta o terminar su generación y forzar que su contribución sea cero. Matemáticamente, consiste en sustituir un estimador unbiased Y por otro estimador unbiased Z, siempre y cuando tengan ambos la misma esperanza. La forma habitual del estimador Z es

$$Z = \begin{cases} \frac{Y}{1-q} & \text{con probabilidad } 1-q \\ 0 & \text{con probabilidad } q \end{cases}$$

donde q es la probabilidad de terminar la ruta. Se puede comprobar fácilmente que Z sigue siendo un estimador unbiased de la integral ya que su esperanza es la misma que la de Y

$$E[Z] = E\left[(1-q)\cdot\frac{Y}{1-q} + q\cdot 0\right] = E[Y]$$

Sin embargo, la varianza del nuevo estimador nunca puede ser menor que la del estimador que sustituye

$$V[Z] = E[Z^2] - E[Z]^2 = \frac{1}{1-q}E[Y^2] - E[Y]^2 \geq V[Y]$$

Por tanto, este método solo está justificado cuando se produce un ahorro en el tiempo de cómputo. La forma en que suele implementarse es aplicando la ruleta rusa en cada punto de rebote. De esta manera, rutas largas parcialmente construidas, que suelen tener menor contribución a la imagen, tienen más probabilidad de ser eliminadas.

1.8. Otras Aproximaciones de la Ecuación de Renderizado

Los artículos que se describen en esta sección no tienen la intención de aproximar completamente la ecuación de renderizado, sino solo ciertas rutas. Turner Whitted [Whi80] no usa métodos estadísticos para resolver ninguna ecuación integral, sino que solo tiene en cuenta los rayos de reflexión y refracción perfecta. Este fue el primer sistema de ray tracing recursivo que recibe el nombre de ray tracing al estilo de Whitted (RTW). Los objetos en las imágenes resultantes tienen la apariencia de espejos perfectos y las sombras producidas no tienen penumbra.

Cook et al. [CPC84] amplían el sistema anterior considerando efectos dispersos (*blured*, en inglés), tales como el desenfoque por movimiento, la profundidad de campo, las penumbras, y las reflexiones y las refracciones difusas. Cada uno de estos efectos se especifica separadamente como una integral sobre un rango de direcciones. La resolución de estas integrales se realiza mediante técnicas de consulta puntuales, de manera parecida a como lo hace Whitted. Este algoritmo recibe el nombre de *ray tracing distribuido* (*distributed ray tracing* o *RTD*). Sin embargo, aunque la cantidad de efectos que se pueden simular es mayor que la conseguida con RTW, no es tan general como el path tracing. Así, algunos efectos no se pueden simular, tales como el *color bleeding* o las *cáusticas*.

Goral et al. [GTGB84] desarrollaron el método conocido como *radiosity*. Primero, la escena se tiene que dividir en un número finito de áreas, llamadas *patches*. Posteriormente, se calcula el porcentaje de intercambio de energía entre todas las parejas de patches, llamado *form factor*. Si todas las superficies son diffuse y uniformes, entonces el transporte de energía desde los patches emisores de luz hacia el resto de patches es equivalente a la resolución de un sistema de ecuaciones lineales. Por último, cualquier método de resolución de ecuaciones se puede usar para obtener la energía emisora de cada patch. Esa energía se usa como color uniforme de cada patch durante el renderizado.

En Pharr y Humphreys [PH10], pág. 926, se muestra la técnica de *Ambient Occlusion* (AO). Esta técnica consiste en, dado un punto x de la escena visible desde la cámara, aproximar por Monte Carlo la siguiente integral

$$\frac{1}{\pi}\int_{\omega\in\mathcal{H}_x} V(x,d,\omega)\,|N_x\cdot\omega|\;d\sigma_x(\omega)$$

donde $V(x,d,\omega)$ indica la visibilidad desde x en la dirección ω a una distancia d. Esta función devuelve 1 cuando el rayo que tiene a x por origen y ω por dirección no encuentra ningún objeto en una distancia d, y 0 en otro caso.

La técnica de AO no es una buena aproximación de la ecuación de renderizado. Sin embargo, las imágenes que produce permiten una rápida apreciación de las formas de los objetos. Esto se debe a que la integral anterior tiene un valor más cercano a cero —y, por tanto, un color más oscuro— en los recovecos de los objetos, y un valor cercano a uno —y, por tanto, un color más claro— en las zonas planas, lo que permite identificar rápidamente esquinas y zonas cóncavas.

H. W. Jensen [Jen96] propone un algoritmo de dos pasadas para aproximar el proceso de difusión. En la primera pasada, el flujo de salida de las luminarias es discretizado en paquetes a los que llama *fotones*. Estos fotones se emiten desde las fuentes de luz y tienen el mismo comportamiento que un haz de luz. En la posición donde se producen las intersecciones, se guarda esa información junto con la dirección de entrada de cada fotón. En la siguiente pasada, los N fotones más cercanos en un punto se usan para aproximar la radiancia de reflexión en ese punto

$$L(x\to\omega_o)\approx\sum_{p=1}^{N} f_r(\omega_o,x,\omega_p)\frac{\Phi_p(x\leftarrow\omega_p)}{\pi r^2}$$

donde Φ_p es el flujo que representa cada fotón, y r es el radio del circulo que encierra los N fotones más cercanos a x.

Hachisuka et al. [HPJ12] presentan un espacio de rutas ampliado en el que rutas que posean puntos diferentes sobre una misma superficie pueden tener contribución diferente de cero a la imagen final. Este nuevo espacio de rutas les permite combinar los algoritmos de photon mapping y BPT. Así, en una primera pasada, el renderizador usa BPT para muestrear rutas. Posteriormente, se renderiza con photon mapping, pero usando como fotones los puntos de las rutas de luz, obtenidos en la pasada anterior.

Capítulo 2

Unidad de Procesamiento Gráfico

¿Qué es lo que fue? Lo mismo que será.
¿Qué es lo que ha sido hecho? Lo mismo que se hará;
y nada hay nuevo bajo del sol. [1]
Eclesiastés 1:9

2.1. Introducción

Las *Unidades de Procesamiento Gráfico* o *GPUs* (de *Graphics Processing Units*) han sido coprocesadores tradicionalmente encargados de implementar en hardware las etapas que componen el algoritmo de la *tubería gráfica* [AMHH08]. La arquitectura actual de las GPUs como hardware paralelo completamente programable ha sido motivada por este algoritmo. Aquí vamos a presentar una breve introducción histórica de las GPUs. Una introducción más completa puede encontrarse en el libro de Kirk y Hwu [KH10].

La tubería gráfica es un algoritmo que se encarga de transformar los triángulos que componen una escena en una imagen bidimensional. Los triángulos de entrada están definidos por sus tres vértices y son enviados por la aplicación a la GPU (de uno en uno o por lotes). La imagen de salida consiste en un array bidimensional de colores (y quizás también otros datos, como *profundidad* o *vectores normales*) resultante de la proyección de esos triángulos sobre la película de la cámara.

Desde que entran los vértices de los triángulos a la tubería gráfica hasta la formación de la imagen final los datos son transformados en una serie de fases que se ejecutan secuencialmente. Estas fases se agrupan principalmente en dos etapas: *la etapa de vértices* y *la etapa de raster*. La función de la etapa de vértices consiste en determinar la posición final de cada vértice en la pantalla multiplicando su posición inicial por determinadas matrices afines y de proyección. Además, se puede añadir una normal por vértice que, junto con la posición de la luz, permiten aplicar un algoritmo de iluminación y obtener un color para cada vértice. La etapa de raster se encarga de determinar el color final de los píxeles que ocupan los triángulos en la imagen mediante interpolaciones de la posición y del color de sus vértices, ambos datos provenientes de la etapa anterior. Debido a que esta etapa es la que da color a cada píxel, a la tubería gráfica se le llama también algoritmo de *rasterizado*.

Desde principios de los años 80 hasta finales de los 90, todas las etapas de la tubería gráfica se han ido implementando en un hardware dedicado conocido como Unidad de Procesamiento Gráfico. Sin embargo, la funcionalidad de esas etapas era fija y solo podía ser escasamente personalizada a través de ciertos parámetros.

Con el tiempo, se buscaron nuevos efectos visuales que pudieran ser implementados con la

[1] La frase "No hay nada nuevo bajo el sol" la repetía frecuentemente el profesor Wen-mei W. Hwu en la escuela de verano PUMPS 2011 en Barcelona durante sus clases. Este capítulo da pruebas de que esa frase es más que cierta.

tubería gráfica. Así, la etapa de vértices se volvió cada vez más programable y cada aplicación podía colocar su propio programa en esa etapa. Estos programas, encargados de modificar los vértices de los triángulos, reciben el nombre de *vertex shaders*. Posteriormente, la etapa de raster también se volvió programable, pudiéndose introducir programas que manipulaban la información asociada con cada píxel[2]. A estos programas se les llama *pixel shaders* o *fragment shaders*. Así, en 2001, la GPU GeForce 3 de NVidia presentó estas características y las series posteriores GeForce 6 y 7 continuaron con su desarrollo. A pesar del aumento de la programabilidad, cada una de estas dos etapas eran implementadas por un hardware diferente dentro del chip.

Con la llegada, en 2006, de la GeForce 8800 se presentó la que se conoce como *arquitectura unificada*, es decir, las etapas de vértices y de raster se ejecutan sobre las mismas unidades funcionales. Además, algunos detalles de la arquitectura paralela subyacente quedaron expuestos para que los programadores tuvieran más control sobre el hardware. Para la programación de este hardware, se presentó un lenguaje de alto nivel, similar a C. A este conjunto de características se le llamó *CUDA* (de *Compute Unified Device Architecture*), aunque es habitual usar este nombre para referirse solo al lenguaje de programación.

2.2. Arquitectura

2.2.1. Generalidades

Las característica de la tubería gráfica anteriormente citadas motivaron el desarrollo de la arquitectura de las GPUs. Primero, la posición y el color de cada vértice no dependen de otros vértices, es decir, todos los vértices procesados en la etapa de vértices son independientes entre sí. De esa forma, un programador escribe un vertex shader solo para un vértice, y ese programa se aplicará a todos de forma independiente. Estas dos características se conocen como independencia de datos y ha motivado la arquitectura SIMD (*Single Instruction on Multiple Data*) de las GPUs. En estas arquitecturas, una serie de unidades funcionales ejecutan las mismas instrucciones en paralelo sobre diferentes datos. En la etapa de raster ocurre lo mismo que en la etapa de vértices aunque, en este caso, es en los fragmentos donde se produce la independencia de datos. En general, sobre esta etapa se han dedicado más recursos (antes de la arquitectura unificada) ya que el número de fragmentos se espera que sea superior al número de vértices.

Segundo, el *framebuffer* es una región de memoria donde se guarda la imagen renderizada por la tubería gráfica. El framebuffer está formado por varios buffers, entre los que se encuentran el *color buffer* (donde se guarda el color de los píxeles de la imagen) o el *z-buffer* (donde se guarda la profundidad de cada píxel). El framebuffer se suele leer o escribir cuando un triángulo es rasterizado, siguiendo el modelo de flujo de datos o *streaming*. Esto implica que los accesos a memoria suelen realizarse en posiciones contiguas y que los datos no son posteriormente (de manera inmediata) reusados. Esto motivó la incorporación a las tarjetas gráficas de sistemas de memoria con mayores anchos de banda que los de las CPUs, en las que el intercambio de datos se realiza de manera transaccional.

Tercero, la independencia de datos también motivó el uso del *multithreading* para ocultar la *latencia* de memoria. La latencia es el tiempo, en ciclos de reloj, que tardan los datos en estar disponibles cuando se realiza una petición a memoria. Entre las memorias que tienen las GPUs, la más lenta es aquella que se encuentra fuera del propio chip (memoria *off-chip*), llamada *memoria global*. Por tanto, cuando un hilo realiza una petición a memoria global va a quedarse muchos ciclos de reloj esperando a que el dato llegue. Durante ese tiempo, otros hilos se despachan y se ejecutan en el procesador, evitando que se quede parado. Si en ningún momento el procesador se queda

[2]El conjunto de información asociado a cada píxel, entre el que se puede encontrar su color, su valor de profundidad o su valor de transparencia *alpha*, se conoce como *fragmento*.

Figura 2.1: Esquema de un SM de la GPU de cap. 2.0.

parado, entonces se dice que la latencia de memoria se ha *ocultado*.

Cuarto, el uso de texturas, a diferencia del framebuffer, sí tiene una buena localidad temporal, es decir, que si se consulta un téxel de una textura, es muy probable que se vuelva a consultar posteriormente. Esto ha motivado el uso de cachés de solo lectura para las texturas.

2.2.2. Detalles

La arquitectura de las GPUs ha cambiado desde que NVidia presentó la primera GPU con CUDA: la GeForce 8800. En el momento de escribir esta tesis, la última arquitectura de las GPUs recibe el nombre de *Fermi* [Nvi09]. Por tanto, mientras no se diga lo contrario, los detalles de la arquitectura de las GPUs que se darán en esta sección corresponden con esta arquitectura. Sin embargo, recientemente han aparecido tarjetas con una nueva arquitectura, llamada *Kepler*.

Las arquitecturas de las tarjetas se designan con una pareja de números de versión, conocidos por *computer capability* (abreviada como *cap.*). La primera cap. en aparecer fue la 1.0 y las GPUs con arquitectura Fermi corresponden con las caps. 2.x (ver [She10] para más detalles de esta arquitectura). En el momento de escribir estas páginas la arquitectura más moderna es la 3.5 [NVia], que corresponde con Kepler. En la guía oficial de programación de NVidia [NVi11] se pueden consultar los detalles de todas las versiones de las arquitecturas de sus GPUs.

Una GPU está formada por una serie de *multiprocesadores* o *SMs* (de *Streaming Multiprocessors*), variando su número de un modelo a otro. Cada SM está formado principalmente por un planificador de hilos, un banco de registros, una memoria compartida, una caché de solo lectura y un array de unidades funcionales (figura 2.1).

El array de unidades funcionales está compuesto por una serie de 32 *cores* en cap. 2.0 y 48 en cap. 2.1. Cada core está formado por unidades aritmético/lógicas de enteros y de reales en coma flotante. Además, existen 16 unidades de lectura y escritura en memoria (LD/ST) y 4 unidades especiales (SFU), encargadas de realizar funciones matemáticas tales como senos o raíces cuadradas. El número de cores aumenta con cada cap. siendo, por ejemplo, de 8 en cap. 1.0, 32 en cap. 2.0 y 192 en cap. 3.0.

El banco de registros mantiene 32K (= 32768) registros de 32 bits. Cada registro es una memoria de acceso rápido *on-chip* donde se guarda información local de cada hilo. Un registro no

puede ser compartido por varios hilos y no pertenece a ningún espacio de direcciones (no se puede acceder a un registro mediante una dirección).

La memoria compartida es también una memoria on-chip. Su velocidad de acceso es mucho mayor que la memoria global, aunque algo menor que la de un registro. Como su propio nombre indica, es común a varios hilos y su función es la de intercambiar información entre ellos. La memoria compartida está formada por varios *bancos* de memoria. Los datos en memoria compartida están repartidos entre estos bancos de manera que secciones de 32 bits consecutivos caen en bancos también consecutivos. Cada banco solo puede proporcionar un dato de 32 bits cada vez, por lo que, en caso de que varios hilos accedan al mismo banco con diferentes direcciones, sus peticiones se serializarán. La cap. 1.x posee 16KB de memoria compartida repartidos en 16 bancos, mientras que en cap. 2.x y 3.x se dispone de hasta 48KB, organizados en 32 bancos.

El planificador de hilos se encarga del manejo de los hilos residentes en cada multiprocesador. Todos los hilos se agrupan en lotes de 32 hilos, llamados *warps*[3]. Siempre se tiene la seguridad de que las operaciones que realizan los 32 hilos de un warp se ejecutan como si se realizasen a la vez[4]. El número máximo de hilos que pueden residir en un multiprocesador en cap. 2.0 es de 1536 (= 48 warps).

Durante la ejecución de un warp, es posible que la siguiente instrucción de alguno de sus hilos no esté lista para ejecutarse. Esto sucede si, por ejemplo, existe algún riesgo de lectura después de escritura sobre un registro, si un dato no está disponible porque todavía no ha llegado de memoria, o si el warp está esperando a los otros warps en una barrera. Si un hilo posee operaciones listas para ser ejecutadas recibe el nombre de hilo *activo*. Si todos los hilos de un warp están activo, ese warp se llama también activo. El planificador se encarga de parar la ejecución de los warps no activos y de despachar un warp activo de entre los disponibles. Como se ha comentado antes, a esta técnica se le conoce como multithreading.

La transferencia de datos con memoria global se realiza siempre mediante lotes llamados *transacciones*. Según el patrón de acceso, en las cap. 1.2 y 1.3 esas transacciones son de 32, 64 o 128 bytes. A partir de la cap. 2.0, las peticiones a memoria global se realizan a través de dos cachés. La caché L1 tiene un tamaño de 48KB como máximo y cada SM posee una. La caché L2 es de 768KB y es única en todo el chip, es decir, que es común para todos los SMs. Esto permite que datos previamente consultados por otros SMs ya residan en caché. A partir de cap. 2.0, las transacciones son de 32 o de 128 bytes, dependiendo de si, respectivamente, la caché L1 está deshabilitada o no.

Existen también otras cachés en las GPUs, aunque son de solo lectura. En concreto la memoria global puede ser leída a través de la caché de textura. En cap. 1.x existe una caché de textura compartida cada 2 o 3 SMs, mientras que a partir de cap. 2.0 cada SM posee su propia caché.

2.3. Modelo de Programación

Antes de la presentación de la arquitectura CUDA, las GPUs también podían usarse como procesadores paralelos generales de alto rendimiento. Sin embargo, su programación resultaba mucho más complicada ya que tenía que hacerse a través de una API gráfica como, por ejemplo, OpenGL. Esta forma de programar las GPUs supuso el comienzo de una nueva área de investigación conocida como *GPGPU* (de *General-Purpose Computing on Graphics Processing Units*).

La manera clásica de programar en GPGPU consistía en, primeramente, codificar y guardar

[3]La palabra *warp* se traduce en español como "urdimbre". Una urdimbre es un conjunto de hilos que se mantienen paralelos en los telares mientras se teje. A los hilos que entrecruzan los hilos de la urdimbre se les llama *trama* o *weft*, en inglés. Obsérvese la relación entre un warp y un conjunto de hilos que se ejecutan en paralelo.

[4]Las arquitecturas con cap. 1.x, por ejemplo, solo poseen 8 cores por lo que las operaciones de los hilos de un warp no se ejecutan realmente a la vez. Sin embargo, estas operaciones se serializan, dando la apariencia de que se realizan simultáneamente.

los datos de la aplicación en los colores RGB de una textura. Posteriormente, el programa se codificaba en la forma de un pixel shader. Por último, se dibujaba un cuadrado en la ventana que ocupaba tantos píxeles como datos se deseaban procesar. Sobre cada píxel se ejecutaba el píxel shader, realizando las operaciones deseadas y escribiendo los resultados en el framebuffer.

Este método de programación tiene varios inconvenientes. Primero, es necesario conocer la API gráfica para poder codificar los datos correctamente. Por tanto, hay que conocer conceptos provenientes del mundo de los gráficos, tales como "textura" o "fragmento", que pueden no tener nada que ver con el problema sobre el que trabaja la aplicación. Segundo, muchos detalles de la arquitectura no eran conocidos por lo que la optimización del código resultaba imposible. Por último y más importante, no era posible escribir en cualquier parte de la memoria ya que cada fragmento tenía como destino una única posición en el framebuffer.

Bruck et al. [BFH+04] presentaron un lenguaje de alto nivel, BrookGPU [BFH+], orientado a la programación de flujos de datos sobre GPU. El compilador se encargaba de transformar este lenguaje a llamadas de la API gráfica, lo que permitía abstraer los detalles de la programación GPGPU clásica. A pesar de la mejora en la legibilidad del código, todavía existían restricciones tales como la imposibilidad de realizar escrituras en cualquier dirección o el de tener un número de datos de entrada máximo (debido a un número máximo de texturas).

Con la llegada de CUDA, la programación de las GPUs se hizo mucho más fácil. La programación se realiza en un lenguaje muy parecido a C, al que se han añadido algunas etiquetas e instrucciones propias. Además, muchos detalles de la arquitectura se han desvelado y se tiene más control sobre el rendimiento del programa. A los programas escritos y compilados en este lenguaje para las GPUs se les llama *kernels*. La función del programador consiste en escribir el código que ejecutará un único hilo. Cuando se lanza la ejecución de un kernel, se especifica el número de hilos y todos ellos ejecutan el mismo flujo de instrucciones. El proceso restante es gestionado automáticamente por la GPU.

Los hilos se agrupan en una jerarquía de dos niveles. Primero, como ya se ha mencionado, grupos consecutivos de 32 hilos se agrupan en warps, cuyas instrucciones se ejecutan a la vez. Esto permite que no sean necesarias barreras a nivel de warp. Sin embargo, tiene el inconveniente de que las divergencias dentro del código se traducen en paradas[5].

El siguiente nivel en la jerarquía es el *bloque*. La decisión de cuántos warps contiene un bloque es responsabilidad del programador. La idea detrás de los bloques es que sus hilos colaboren entre sí, ya que la memoria compartida y las barreras solo son visibles para los hilos pertenecientes a un mismo bloque. Además, se tiene la garantía de que sus hilos se ejecutan siempre en el mismo SM. De esta forma, los hilos de un bloque pueden intercambiar información o mantener datos globales por bloque a través de la memoria compartida. Al conjunto de todos los bloques y, por tanto, de todos los hilos, se le llama *grid*.

La colaboración entre hilos de diferentes bloques no se puede realizar dentro del mismo kernel debido a la ausencia de barreras globales para todos los hilos del kernel. Sin embargo, terminar un kernel y lanzar otro tiene el mismo efecto que una barrera global, aunque parte del control de la aplicación se tiene que trasladar a CPU, complicándose la programación.

Es en el lanzamiento a ejecución de un kernel cuando se especifica el número de hilos por bloque y el número de bloques por grid. En ese momento, se asignan los recursos de cada SM: los registros se reparten entre los hilos y la memoria compartida entre los bloques. Esto determina el número máximo de bloques que residirán en cada SM simultáneamente, cuyos hilos serán usados

[5]Supongamos, por ejemplo, que un warp va a ejecutar una instrucción *if-then-else*. Supongamos también que algunos hilos de ese warp evalúan la condición a cierto, por lo que deben ejecutar la parte *then*, y el resto a falso, tienen que ejecutar la parte *else*. En este caso, mientras algunos hilos ejecutan la parte *then*, el resto se queda esperando ya que no es posible que hilos del mismo warp ejecuten instrucciones diferentes. Lo mismo ocurre para la parte *else*. El resultado es que se ha dedicado el mismo tiempo que si se hubieran ejecutado las dos ramas secuencialmente.

para multithreading. La relación entre el número de hilos asignados a un multiprocesador y el máximo que permite el hardware recibe el nombre de *ocupación* de la GPU. El caso ideal es aquel con una tasa de ocupación del 100 %, donde cada SM posee el máximo número de hilos para explotar el multithreading. Sin embargo, las necesidades de los programas hacen difícil alcanzar esa tasa en la práctica (Ryoo et al. [RRB+08]).

Es habitual que no existan suficientes recursos en la GPU para ejecutar todos los bloques a la vez. Los bloques que no han obtenido recursos se quedan esperando. Cuando todos los hilos de un bloque terminan su ejecución, sus recursos son liberados y un bloque que estaba esperando se apodera de ellos. En arquitecturas anteriores a Fermi, la asignación de bloques se realiza antes del comienzo de la ejecución (planificación estática por SM), aunque en las GPUs más recientes cada bloque se ejecuta en un SM que quede libre (planificación dinámica).

2.4. Aplicación de la GPU en la Construcción de Primitivas

Una *primitiva paralela* (*data-parallel primitive* en inglés) es una operación sencilla que se realiza sobre un array de datos en un hardware paralelo. Las primitivas se usan como bloques de construcción para implementar algoritmos más complejos. La ventaja del uso de primitivas en la construcción de programas en GPU es doble. Por un lado, se dispone de implementaciones rápidas específicamente diseñadas y optimizadas para las GPUs. Por otro, el programador no necesita aprender los detalles particulares de la programación de las GPUs, le basta invocar secuencialmente las primitivas paralelas adecuadas desde la CPU.

2.4.1. Scan

Una de las primitivas más empleadas en la construcción de programas paralelos es la operación conocida como *suma de prefijos* o *scan*. La función scan recibe como entrada un array de n elementos y una operación binaria $\oplus$. La salida es un array con el mismo número de componentes donde cada elemento es la aplicación de $\oplus$ sobre todos los elementos anteriores. Si en esta aplicación también se considera el propio elemento entonces se le llama *scan inclusivo*, si no, *scan exclusivo*. Para la descripción del scan que vamos a realizar solo es necesario que la operación $\oplus$ sea asociativa y que tenga elemento neutro. En la práctica, las operaciones que se usan son la suma, el producto, el máximo o el mínimo, que además son conmutativas. Por simplicidad, usaremos la operación suma + en los ejemplos, si no se dice lo contrario.

Formalmente, dado el array de entrada de n componentes $a = [a_0, \ldots, a_{n-1}]$ y la operación binaria +, la salida del scan inclusivo es

$$scan_{in}(a) = [a_0,\ (a_0 + a_1),\ \ldots\ ,\ (a_0 + \cdots + a_{n-1})]$$

mientras que la salida del scan exlusivo es

$$scan_{ex}(a) = [0,\ a_0,\ (a_0 + a_1),\ \ldots\ ,\ (a_0 + \cdots + a_{n-2})]$$

Como se puede ver, el scan inclusivo se puede obtener a partir del scan exclusivo, sumándole el array de entrada elemento a elemento.

$$scan_{in}(a) = a + scan_{ex}(a)$$

Existe también una versión *segmentada* de esta operación, donde el array de entrada está dividido en varios segmentos. La suma de los elementos anteriores se realiza solo para los valores que pertenezcan al mismo segmento, ignorando el resto. Por ejemplo, para el siguiente array

$$a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]$$

su scan segmentado inclusivo es

$$scan\_seg_{in}(a) = [[1, 3, 6], [4, 9, 15], [7, 15, 24]]$$

Implementaciones del scan

Las primeras implementaciones del scan no se realizaron sobre las GPUs, sino sobre el hardware paralelo de la época. G. E. Blelloch [Ble90] implementó el scan con un algoritmo de dos fases (*up-down-scan*[6]). La primera, llamada *up-sweep* o de *reducción*, obtiene valores de reducción intermedios en el array. La siguiente fase, llamada *down-sweep*, distribuye los valores de la etapa anterior por el array para obtener el scan exclusivo final. Cada una de estas fases consiste en una serie de etapas ejecutadas secuencialmente. El número de etapas es de orden logarítmico con respecto al tamaño del array. Además, las operaciones de cada etapa son independientes y se pueden realizar en paralelo.

Hillis y Steele [HS86] desarrollan un scan inclusivo en solo una fase (*up-scan*), reduciendo, por tanto, el número de etapas a la mitad. Al igual que Blelloch [Ble90], el número de etapas es del orden logarítmico con respecto al tamaño del array de entrada y las operaciones de cada etapa se pueden ejecutar en paralelo.

Chatterjee et al. [CBZ90] implementan un scan exclusivo sin seguir el esquema recursivo de los dos anteriores. En este caso, los datos se colocan en forma de matriz y cada procesador se encarga secuencialmente de realizar el scan de una columna (*sequential-scan*). Cada columna es independiente de las demás, por lo que su procesamiento se realiza en paralelo. Los resultados parciales de cada columna son posteriormente tratados para obtener el scan final.

D. Horn [Hor05] realizó una de las primeras implementaciones en GPU del scan. El autor implementó el algoritmo up-scan para realizar una compactación (llamada *data filtering* en su artículo). Su implementación se realizó en GPGPU clásica mediante BrookGPU [BFH+04]. Para solucionar el problema de la carencia de escrituras en cualquier posición de memoria, el autor implementó un algoritmo parecido a una búsqueda binaria de forma que cada elemento del array final "recoge" su valor del array previo.

Sengupta et al. [SLO06] implementaron up-down-scan sobre GPUs y lo compararon con la implementación del up-scan de Horn. Obtuvieron mejores resultados porque el algoritmo up-down-scan de Blelloch es más eficiente cuando el número de procesadores de la GPU es menor que el número de elementos de entrada. Además, desarrollaron un algoritmo híbrido consistente en ejecutar el up-down-scan hasta que el número de elementos caiga por debajo del grado de paralelismo de las tarjetas, llamando en ese caso al up-scan de Horn.

Con la llegada de CUDA se pudieron hacer implementaciones más eficientes en GPU de los algoritmos de scan antes citados debido al mayor conocimiento de su arquitectura. Así, Sengupta et al. [SHZO07] implementan el algoritmo up-down-scan, y M. Harris [Har07b] y Harris et al. [HSO07] implementan el mismo algoritmo, pero haciendo hincapié en los conflictos de bancos. Dotsenko et al. [DGS+08] implementan un scan en CUDA al estilo de sequential-scan. Sengupta et al. [SHG08, SHGO11] implementan una versión recursiva del scan, que se apoya en una función que realiza up-scan solo a nivel de warp. Esta última versión se describe a continuación.

Implementación eficiente de scan en CUDA

La implementación del scan que describimos en esta sección se encuentra en la librería CUDPP [HOS+10], cuyo código está basado en el trabajo de Sengupta et al. [SHG08]. El algoritmo está dividido en tres etapas, cada una de las cuales se encarga de realizar el scan sobre una porción cada vez mayor del array de entrada.

[6] El nombre de este algoritmo y de los dos siguientes no es estándar. Se han elegido estos nombres para que la posterior presentación de sus implementaciones en GPU sea más sencilla.

```
1  // s_temp: array de memoria compartida donde se encuentran los datos.
2  template<ScanKind kind>
3  __device__ int scan_warp(volatile int* s_temp){
4      const int thid = threadIdx.x;
5      const int lane = thid % 32;
6
7      // Scan inclusivo de 32 elementos.
8      if(lane >= 1)  s_temp[thid] += s_temp[thid - 1];
9      if(lane >= 2)  s_temp[thid] += s_temp[thid - 2];
10     if(lane >= 4)  s_temp[thid] += s_temp[thid - 4];
11     if(lane >= 8)  s_temp[thid] += s_temp[thid - 8];
12     if(lane >= 16) s_temp[thid] += s_temp[thid - 16];
13
14     // Distinguimos entre inclusivo y exclusivo.
15     if(kind == inclusive) return s_temp[thid];
16     else                  return (lane > 0) ? s_temp[thid-1] : 0;
17 }
```

Figura 2.2: Código de `scan_warp`.

Scan de un warp. La función `scan_warp` se encarga de realizar el scan (inclusivo o exclusivo) sobre 32 elementos consecutivos del array de entrada usando un único warp. La implementación de este algoritmo está basada en el up-scan de Hillis y Steel [HS86]. Primeramente, los datos sobre los que se va a realizar el scan se traen desde memoria global a memoria compartida. Por tanto, y mientras no se diga lo contrario, asumimos que los datos ya se encuentran disponibles en el array `temp` en memoria compartida. Además, suponemos también que cada dato tiene un tamaño de 4 bytes como, por ejemplo, es el caso de los tipos `int` o `float`.

El código CUDA[7] de `scan_warp` se muestra en la figura 2.2. Para realizar el scan sobre 32 elementos, se realiza un bucle de 5 ($= \log_2 32$) pasadas sobre el array `temp`. Ya que el número de vueltas del bucle es conocido, es posible desplegarlo (líneas 8–12) evitando la sobrecarga relacionada con el propio tratamiento del bucle (tales como los saltos y la evaluación de la condición). El número de pasadas que realiza la versión up-down-scan es el doble, aunque el número de operaciones totales es menor. Sin embargo, eso no se traduce en ahorro ya que los hilos de un warp se ejecutan de manera SIMD. Por tanto, es más eficiente disminuir el número de las pasadas que el número de las operaciones totales que se realizan.

En la pasada d-ésima (la primera pasada es $d = 0$), el hilo con identificador `thid` realiza una suma de los elementos que se encuentran en `a[i]` y en `a[i-2^d]`. En el caso de que el índice `i-2^d` haga referencia a un elemento que se encuentra fuera de los límites del array, ese hilo no hace nada. (implementado con los `if`s de las líneas 8–12). Además, no se requieren sincronizaciones explícitas, ni existen conflictos de bancos. En la figura 2.3 se ve un ejemplo de un `scan_warp` sobre un array inicializado con 1's.

Scan de un bloque. Supongamos que un bloque contiene $b = 32 \cdot w$ hilos, donde w es un entero mayor que 1 y que indica el número de warps de un bloque. La función `scan_block` hace uso de la función `scan_warp`, implementada en el apartado anterior, para realizar un scan sobre b elementos consecutivos del array de entrada. La función `scan_block` está dividida en 5 pasos y su código se muestra en la figura 2.4. Para facilitar la descripción del algoritmo, supondremos que un bloque está formado por 32 warps ($w = 32$), es decir, $1024 = 32 \cdot 32$ hilos.

Antes de ejecutar `scan_block`, todos los hilos del bloque traen la sección del array sobre la que van a hacer scan a memoria compartida (guardada en el array `temp`). Lo primero que

[7] El convenio que usaremos para nombrar a las variables es el siguiente: si una variable se encuentra en memoria compartida, entonces comienza por `s_`, si se encuentra en memoria global comienza por `g_`, si es un registro, entonces no lleva ningún prefijo.

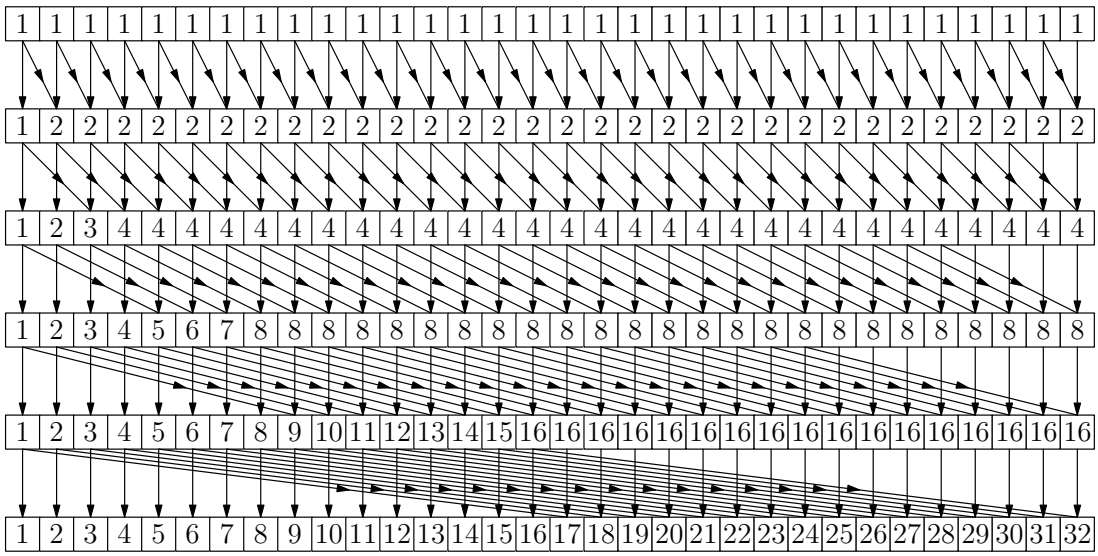


Figura 2.3: Ejemplo de la función `scan_warp` con la operación + sobre enteros. La función `scan_warp` realiza un scan inclusivo sobre 32 elementos usando un warp (= 32 hilos). En la etapa d se suman elementos que se encuentran a una distancia de 2^d.

```
template<ScanKind kind>
__device__ int scan_block(volatile int* s_temp) {
    const int thid = threadIdx.x;
    const int lane = thid & 31;  // lane = thid % 32;
    const int wid  = thid >> 5;  // wid = thid / 32; (warp id)

    // Paso 1.
    int val = scan_warp<kind>(s_temp);
    __syncthreads();

    // Paso 2.
    if(lane == 31) s_temp[wid] = s_temp[thid];
    __syncthreads();

    // Paso 3.
    if(wid == 0) scan_warp<inclusive>(s_temp);
    __syncthreads();

    // Paso 4.
    if(wid > 0) val += s_temp[wid - 1];
    __syncthreads();

    // Paso 5.
    s_temp[thid] = val;
    __syncthreads();

    return val;
}
```

Figura 2.4: Código de `scan_block`.

hace `scan_block` es realizar el scan de cada lote de 32 elementos consecutivos usando `scan_warp` (paso 1). Al terminar el paso 1, cada elemento ha computado la suma de sus elementos anteriores desde el comienzo de su segmento de 32 datos. Por tanto, para completar el scan, es necesario sumar también los datos de los segmentos anteriores, lo que se consigue en dos fases. Primero, se realiza el scan solo sobre los últimos elementos de cada warp (pasos 2 y 3), es decir, en los primeros elementos de `temp` se encuentra ahora el scan de los valores que han resultado al reducir cada segmento. Por último, el resto de elementos solo tienen que sumar a su valor el correspondiente de `temp` para completar su scan (pasos 4 y 5).

Scan completo. Tras haberse realizado `scan_block`, cada elemento del array de entrada tiene su valor de scan correcto dentro de su bloque de b elementos. Al igual que ocurre con `scan_block`, es necesario sumarle a cada elemento los resultados parciales de los bloques anteriores en el array completo.

El código del scan completo sigue un esquema parecido al código en 5 pasos de la figura 2.4. Primero, la llamada a la función `scan_warp` del paso 1 se sustituye por una llamada al kernel `scan_block`, para que se realice el scan de las secuencias de b elementos consecutivos del array de entrada. Segundo, de manera similar al paso 2, se guardan en un nuevo array `block_results` la suma de los b elementos de cada secuencia del array de entrada. Tercero, la función `scan_warp` del paso 3 no se puede sustituir siempre por el kernel `scan_block` ya que es posible que el array `block_results` tenga más de b bloques, sino que tiene que sustituirse por una nueva llamada al scan completo. Esto convierte al scan completo en una función recursiva cuyo caso base se alcanza cuando se tiene que realizar el scan de un único bloque de b elementos mediante el kernel `scan_block`. Cuarto, a la vuelta de la recursión y de manera similar a los pasos 4 y 5, cada elemento de cada secuencia acumula su valor correspondiente del array `block_results` dando como resultado el scan completo del array de entrada. Obsérvese que las sincronizaciones explícitas que se encuentran después de cada paso del algoritmo de la figura 2.4 desaparecen porque la ejecución de los kernels se controla desde la CPU, lo que se entiende como una sincronización implícita de todos los hilos del grid (sección 2.3).

2.4.2. Compactación

Dado un array de entrada IN y una propiedad P, la salida de esta primitiva es un array que solo contiene aquellos elementos de IN que cumplen P. Esta función también recibe el nombre de *filtro*, ya que elimina de la salida los elementos que no cumplen P. La implementación en GPU de la primitiva *compactación* se realiza en dos fases. A continuación se muestra un ejemplo sencillo:

a	**b**	**c**	**d**	**e**	**f**	**g**	IN
1	0	1	0	0	1	1	$e = P(IN)$
0	1	**1**	2	2	**2**	**3**	$address = scan_{ex}(e)$
						4	$NT = e[6] + address[6]$
a	**c**	**f**	**g**				OUT

En la primera fase, se evalúa P para cada elemento en paralelo y su resultado se guarda en el array e. Posteriormente, se realiza un scan exclusivo sobre e para obtener la dirección final de cada elemento. Ya que el resultado de evaluar P está guardado en e con valores 1 (cierto) o 0 (falso), tras la ejecución del scan, cada elemento que cumple P sabe cuántos elementos tiene a su izquierda. Eso permite una escritura en paralelo de los elementos ciertos al array OUT sin colisiones. El número NT de elementos que cumplen P se puede conocer a partir del último valor de e y del scan.

2.4.3. Split

Esta primitiva es muy parecida a la compactación. En este caso, los elementos de la entrada *IN* que cumplen la propiedad P se guardan al final del array y los que no la cumplen se guardan al principio, manteniéndose el orden relativo entre elementos del mismo grupo. Al igual que en la compactación, la dirección de cada elemento en el array de salida se puede obtener realizando un scan exclusivo sobre el resultado de la evaluación de la propiedad P, como se puede ver en el siguiente ejemplo:

a	**b**	**c**	**d**	**e**	**f**	**g**	IN
1	0	1	0	0	1	1	$P(IN)$
0	1	0	1	1	0	0	$e = \neg P(IN)$
0	**0**	1	**1**	**2**	3	3	$f = scan_{ex}(e)$
						3	$NF = f[6] + e[6]$
0	1	2	3	4	5	6	id de cada hilo
3	4	**4**	5	5	**5**	**6**	$t = id - f + NF$
3	0	4	1	2	5	6	$address = e\ ?\ f\ :\ t$
b	**d**	**e**	**a**	**c**	**f**	**g**	OUT

Primeramente, se evalúa la propiedad P en paralelo sobre cada elemento, escribiendo en el array e los valores 1 (falso) o 0 (cierto). El scan exclusivo de e ya obtiene la dirección final en el array de salida de los elementos falsos (array f, en negrita), análogamente a lo que sucedía con la compactación. La dirección de cada elemento que sí cumple P se calcula a través de su índice sobre el array (id), y eliminando el número de elementos falsos a su izquierda (guardado en $f[id]$). Eso resulta en la posición relativa de los elementos ciertos. Para obtener su dirección final (array t, en negrita) solo es necesario sumar a su dirección el número total de elementos falsos (guardado en NF). Por último, cada hilo escribe con seguridad su elemento en el array destino sin que existan colisiones.

2.4.4. Radix-sort

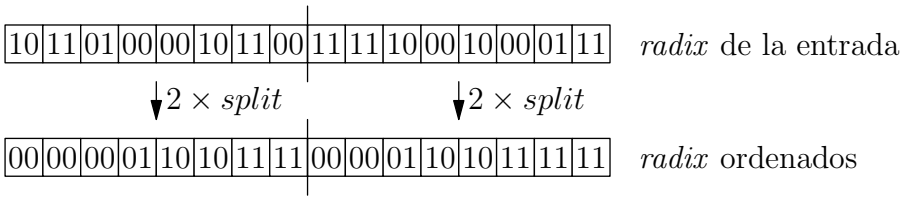
Para ordenar un array por radix-sort vamos a suponer que los bits de cada dato se agrupan en secciones de r bits, llamados *radix*. En cada iteración, radix-sort ordena los elementos en función de su radix, desde el radix menos significativo hasta el más significativo[8], usando una ordenación por *counting-sort*, como se describirá a continuación. Para simplificar la descripción, supondremos que el array está formado por enteros positivos.

Ya que la primitiva split es capaz de ordenar un array de enteros en función de un único bit, una implementación directa de un 2-radix-sort en paralelo (Blelloch [Ble90]) consiste en ejecutar la primitiva split tantas veces como bits tiene cada entero, desde el menos hasta el más significativo. Sin embargo, como señalan Satish et al. [SHG09], esta implementación directa es demasiado lenta en GPU ya que cada split tiene que leer y escribir datos en memoria global. Por tanto, estos autores proponen una implementación alternativa, basada en un trabajo de Zagha y Blelloch [ZB91], en la que secciones del array se ordenan en memoria compartida mediante split y los resultados son globalmente mezclados con counting-sort. Así, el algoritmo en CUDA del radix-sort consta de cinco pasos secuenciales que ordenan el array de entrada por un único radix de r bits. Si se desea ordenar el array completo es necesario ejecutar estos cinco pasos iterativamente desde el

[8]Como el algoritmo ordena un radix en cada iteración y cada radix tiene 2^r valores posibles, el algoritmo recibe el nombre de 2^r-radix-sort.

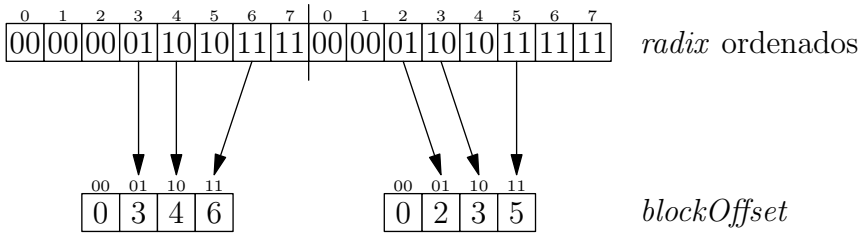
radix menos significativo hasta el más significativo, tantas veces como sea necesario. Los pasos del algoritmo se detallan a continuación.

Paso 1: Ordenación de cada bloque. Cada bloque de hilos carga desde memoria global a memoria compartida tantos enteros como hilos tiene ese bloque. Posteriormente, se realizan r operaciones split que dejan cada sección ordenada. En la figura siguiente se puede ver un ejemplo sencillo con $r = 2$ bits y 8 hilos por bloque:



La salida de este paso son los radix de la entrada ordenados por cada bloque, es decir, la salida de un 2^r-radix-sort.

Paso 2: Actualización de *blockOffset*. Al llegar a este paso, los radix de los elementos de cada sección están consecutivos y ordenados. En este paso, se guarda, por bloque, el índice de comienzo de cada conjunto de valores con el mismo radix



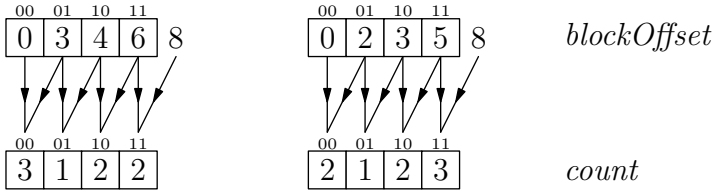
El array donde se guarda este índice se llama *blockOffset* y tiene 2^r elementos. Como en nuestro ejemplo tenemos cuatro posibles radix ($2^r = 4$), el tamaño de blockOffset es de 4. Primero, el array blockOffset se inicia a cero en paralelo. Los primeros hilos del bloque se encargan de ejecutar esta tarea:

```
1 if(thid < 4) {
2     s_blockOffset[thid] = 0;
3 }
```

Posteriormente, cada uno de los 8 hilos compara su radix con el anterior. Si son diferentes, entonces ese elemento es el comienzo de un conjunto de elementos con el mismo radix. Así, su hilo guarda su índice en blockOffset:

```
1 if(thid > 0 && s_radix[thid] != s_radix[thid-1]) {
2     s_blockOffset[s_radix[thid]] = thid;
3 }
```

Paso 3: Actualización del array *count*. El array *count* indica el número de elementos con el mismo radix para cada bloque



Primeramente, hay que iniciar el array count a cero, a igual que se hizo con el array blockOffset:

```
if(thid < 4) {
    g_count[thid] = 0;
}
```

Los valores de este array se obtienen a partir de blockOffset, calculando la distancia entre el comienzo de un segmento y el anterior:

```
if(thid > 0 && s_radix[thid] != s_radix[thid-1]) {
    g_count[radix[thid-1]] = thid - s_blockOffset[s_radix[thid-1]];
}
```

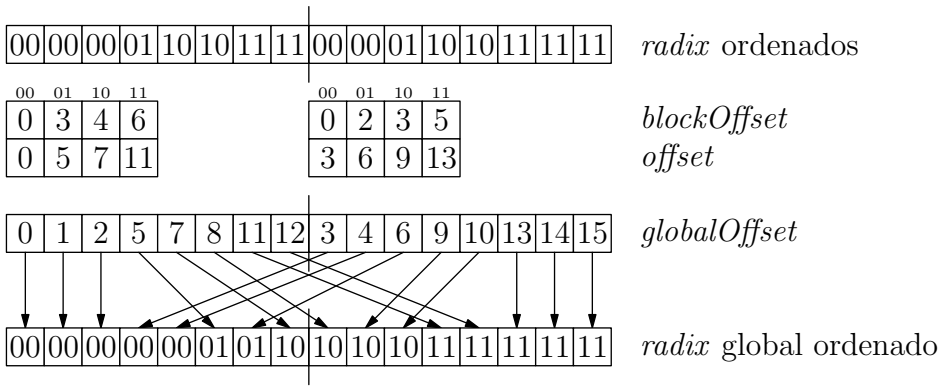
El último elemento de *count* es un caso especial ya que no tiene ningún elemento a su derecha para actualizarlo. Por tanto, es el último hilo el que se encarga de hacerlo:

```
if(thid == 7) {
    g_count[radix[7]] = 8 - s_blockOffset[radix[7]];
}
```

Paso 4: Actualización del array *offset*. El array *offset* indica el número de datos con radix menor o igual que existen a la izquierda de cada elemento, sobre el array total. Este array se obtiene guardando los arrays count de cada bloque en memoria global de manera que los elementos referentes al mismo radix queden consecutivos. Posteriormente, se realiza un scan exclusivo sobre los datos anteriores. Una manera equivalente de ver esta operación es considerar que cada array count forma una fila de una matriz y el scan exclusivo se realiza por columnas, como se puede ver a continuación:

Paso 5: Posición final de cada elemento. A partir de los arrays blockOffset y offset se puede calcular la dirección final que tendrá cada elemento en memoria global después de la ordenación de un radix:

```
segmentOffset = thid - s_blockOffset[s_radix[thid]];
globalOffset = g_offset[s_radix[thid]] + segmentOffset;
```



Este algoritmo ha sido implementado en la librería CUDPP [HOS+10], cuya configuración usa bloques formados por 256 hilos y cada radix posee 4 bits ($r = 4$, que corresponde a un 16-radix-sort).

2.5. Reducción

La primitiva paralela *reducción* se encarga de aplicar la operación $\oplus$ a un conjunto de valores del array de datos de entrada. Al igual que sucede con scan, existen dos variedades de esta primitiva: *reducción no segmentada* y *reducción segmentada*. Esta primitiva se presenta en una sección independiente de la sección 2.4 porque corresponde a nuestro trabajo de Martín et al. [MATG12].

2.5.1. Reducción No Segmentada

La primitiva reducción no segmentada, *red*, consiste en aplicar una operación binaria $\oplus$ a un array de entrada de n componentes, $a = [a_0, \cdots, a_{n-1}]$, de la siguiente manera

$$red(a) = a_0 \oplus \cdots \oplus a_{n-1}$$

Si asumimos que la operación $\oplus$ es asociativa, entonces el resultado de la reducción no segmentada es único. En el código, nos referiremos a la operación $\oplus$ como `op`.

La implementación en CUDA de la reducción no segmentada se realiza de tal forma que cada bloque de B hilos reduce una sección del array de entrada compuesta por D elementos, a la que llamaremos *bloque de datos* (los valores concretos de B y D dependen de la implementación y se verán más adelante). Si el array de entrada tiene más elementos que D, entonces el kernel se lanza con más de un bloque de hilos. Cada uno de ellos reduce su bloque de datos asignado y proporciona un único valor, por lo que la salida de ese kernel consiste en un array con tantos elementos como bloques de hilos se han ejecutado. Para obtener el valor final de la reducción, se tiene que ejecutar nuevamente este kernel para reducir la salida de la ejecución anterior. Así se procede hasta que la salida tenga un único elemento, siendo este el resultado final. Este enfoque recursivo recibe el nombre de *multipasada*.

Es posible que el tamaño de la entrada no sea múltiplo de D, en cuyo caso el último bloque reduce menos elementos que los demás. Existen dos maneras de solventar esta diferencia sin modificar la forma en que se realiza la reducción: *padding* y *virtual padding*. El padding consiste en aumentar el array para que su tamaño sea múltiplo de D. Los elementos extra se rellenan con el elemento neutro $1_\oplus$ para que no afecten al resultado final. El virtual padding consiste en modificar la etapa de carga. Así, si un hilo tiene que cargar datos que están fuera del límite del array, entonces no leerá de memoria global sino que escribirá $1_\oplus$ en memoria compartida. Esta última versión es la que hemos usado ya que no consume más memoria global y es fácil de implementar.

La ejecución de cada kernel se divide en tres etapas. En la primera, los hilos traen su bloque de datos desde memoria global a memoria compartida. En la siguiente etapa, los hilos de cada bloque se encargan de obtener el valor de reducción operando sobre esos datos. En la etapa final, el resultado se guarda en memoria global. En esta sección solo nos vamos a centrar en la segunda etapa, es decir, aquella en la que los hilos reducen su bloque de datos. Por tanto, en lo sucesivo asumiremos que los datos de entrada se encuentran disponibles en memoria compartida.

Reducciones tree-based

Existen dos enfoques para implementar la reducción en CUDA: el basado en árbol (o *tree-based*) y el secuencial. Los algoritmos tree-based están basados en la exploración de un árbol cuyas hojas poseen la información del array de entrada y cuyos nodos internos se evalúan según la siguiente ecuación recursiva

$$red(n) = \begin{cases} red(n.l) \oplus red(n.r) & \text{si } n \text{ es nodo interno} \\ value(n) & \text{si } n \text{ es una hoja} \end{cases} \tag{2.1}$$

```
1  void TB2_reduction (float* s_data){
2    // ID del hilo.
3    unsigned int thid = threadIdx.x;
4    // Distancia entre los datos.
5    unsigned int stride = 1;
6    // Indices de los valores de reducción.
7    unsigned int i, ai, bi;
8    // Bucle de la reducción. El numero de pasadas es log2(B)+1.
9    for(unsigned int d = B; d > 0; d >>= 1) {
10     __syncthreads();
11     // Los d primeros hilos reducen.
12     if(thid < d){
13       // Indices de los valores que se reducen.
14       i = 2 * stride * thid;
15       ai  = i;
16       bi  = ai + stride;
17       // Evitar conflictos de banco.
18       ai += (ai >> 5); // log2(numBancos)=log2(32)=5
19       bi += (bi >> 5); // log2(numBancos)=log2(32)=5
20
21       // Reducción de dos elementos.
22       s_data[ai] = op(s_data[ai], s_data[bi]);
23     }
24     stride <<= 1; // stride *= 2;
25   }
26   // La reducción del bloque se queda guardada en s_data[0].
27 }
```

Figura 2.5: Algoritmo de reducción del kernel de TB2. La constante `numBancos` indica el número de bancos en que está dividida la memoria compartida, que es 32 en la GPU que hemos usado.

donde *value* devuelve un valor de la entrada, y $n.l$ y $n.r$ son los hijos izquierdo y derecho del nodo interno n, respectivamente. Este árbol, sin embargo, no necesita construirse realmente ya que el único valor que interesa es el valor de reducción completo, que se encuentra en $red(root)$. Además, se usa el propio array de entrada para guardar la información de los nodos intermedios y simular iterativamente el carácter recursivo de este algoritmo. Hemos realizado dos implementaciones paralelas en CUDA de la reducción tree-based: *TB2* y *TB4-warp*. Estos algoritmos se encuentran en nuestro trabajo de Martín et al. [MATG12].

Reducción TB2. El código del algoritmo de reducción TB2 se muestra en la figura 2.5 y un ejemplo de su ejecución se muestra en la figura 2.6. Este algoritmo está basado en el algoritmo *reduction#2* de M. Harris [Har07a]. En cada iteración, cada hilo activo reduce dos elementos previamente reducidos en la iteración anterior (línea 22). Por tanto, el número total de iteraciones del bucle `for` (línea 9) es $\log_2 D = \log_2 B + 1$, donde $B = 256$ y $D = 512$ en nuestra implementación. El número de hilos activos en cada iteración es la mitad que la anterior (línea 12), terminando en un único hilo activo en la última iteración. La salida de ese último hilo es el valor de reducción total, que se guarda en la primera componente de `s_data`. El número de warps activos también se reduce a la mitad en cada iteración, salvo en las seis últimas iteraciones, en las que solo hay un warp activo.

Los dos índices de los elementos que se reducen, `ai` y `bi`, se obtienen a partir del valor de desplazamiento `stride` (líneas 15–16), que indica la distancia entre ellos. Esta distancia comienza en 1 (línea 5) y se dobla en cada iteración (línea 24). El resultado de la reducción se escribe sobre el elemento de índice `ai`, lo que llamamos *left-storing*. Una versión alternativa hubiera sido guardar el elemento en `bi`, lo que llamamos *right-storing*. Para evitar conflictos de bancos, es suficiente con añadir un elemento extra cada 32 elementos del bloque de datos. Ese elemento extra, o *padding*,

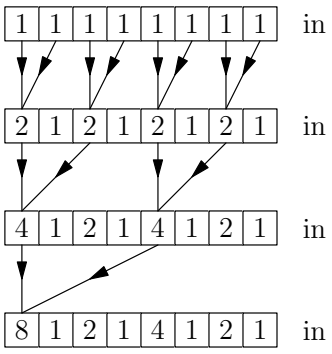


Figura 2.6: Ejemplo sencillo de reducción TB2 con $B = 4$ y $D = 8$. En el primer elemento del array, in[0], se encuentra la reducción de toda la entrada. El primer hilo se encarga de guardar ese elemento en memoria global.

no guarda información del array de entrada ni se consulta durante la reducción, aunque se tiene que tener en cuenta en los cálculos de `ai` y `bi` (líneas 18–19).

Reducción TB4-warp. El algoritmo TB2 tiene el inconveniente de que ejecuta una sincronización en cada iteración del bucle, lo que produce un gran impacto en el rendimiento (sección 2.2.2). Para reducir el uso de estas barreras, hemos implementado el algoritmo TB4-warp (figura 2.7). En este algoritmo se aplica la misma técnica empleada por Sengupta et al. [SHG08] para scan (sección 2.4.1), que consiste en utilizar como barrera la sincronización implícita de los hilos de un warp. Así, implementamos la función `tb_reduceWarp` (línea 2), que es una adaptación left-storing de la función `scan_warp` (figura 2.2). Esta función se encarga de reducir una secuencia de 32 datos consecutivos sin el uso de barreras, y se usa como base para implementar la reducción de un bloque de datos.

Primeramente, secuencias de 32 datos del bloque de datos se reducen usando `tb_reduceWarp` (líneas 18–28). Posteriormente, los resultados tienen que reducirse de nuevo para obtener el valor de reducción final de todo el bloque de datos. Esta nueva reducción se realiza también con una ejecución de `tb_reduceWarp` (línea 32), tras haber realizado una sincronización de los warps anteriores. Siguiendo este esquema, el número máximo de elementos que se pueden reducir es de $D = 32 \cdot 32 = 1024$. Para alcanzar este tamaño, y debido a que el tamaño elegido para el bloque CUDA es de $B = 256$ hilos, cada warp debe reducir secuencialmente 4 lotes de 32 elementos (línea 24).

El algoritmo TB4-warp tiene dos ventajas con respecto a TB2. La primera es que las barreras han disminuido a solo una (línea 28), por lo que se disponen de más warps activos para realizar multithreading. La segunda es la ausencia de conflictos de banco debido a que `tb_reduceWarp`, que es la función base del procedimiento, está libre de ellos. Sin embargo, tiene el inconveniente de que el tiempo que precisa cada warp es mayor que en TB2, ya que siempre existe un porcentaje de hilos en cada warp que están activos (sentencias `if` en la función `tb_reduceWarp`).

Reducción secuencial

La reducción secuencial se basa en que cada hilo reduce iterativamente una secuencia de datos en lugar de hacerlo recursivamente, como hacen los tree-based. El algoritmo que implementa la reducción secuencial recibe el nombre de *Matrix* y también se encuentra en nuestro trabajo de Martín et al. [MATG12].

Matrix. El código de este algoritmo se muestra en la figura 2.8 y está basado en el código del

```
1  // Reducción intra-warp.
2  void tb_reduceWarp(float* s_data, unsigned int thid,
3                     unsigned int lane, float& target){
4    if(!(lane & 1))                                          // if(lane%2==0)
5          s_data[thid] = op(s_data[thid], s_data[thid+1]);
6    if(!(lane & 3))                                          // if(lane%4==0)
7          s_data[thid] = op(s_data[thid], s_data[thid+2]);
8    if(!(lane & 7))                                          // if(lane%8==0)
9          s_data[thid] = op(s_data[thid], s_data[thid+4]);
10   if(!(lane & 15))                                         // if(lane%16==0)
11         s_data[thid] = op(s_data[thid], s_data[thid+8]);
12   if(!(lane & 31))                                         // if(lane%32==0)
13             target = op(s_data[thid], s_data[thid+16]);
14 }
15
16 // Reducción TB4-warp.
17 void TB4_warp_reduction (float* s_data){
18   __shared__ float s_result[32];     // 32 resultados parciales
19   unsigned int thid   = threadIdx.x; // ID del hilo
20   unsigned int warpid = thid >> 5;   // warpid = thid/32
21   unsigned int lane   = thid & 31;   // lane = thid%32
22
23   // Reducción de una secuencia de 128 datos.
24   for(unsigned int k = 0; k < 4; k++) {
25     // Reduccion intra-warp
26     tb_reduceWarp(s_data, thid+k*blockDim.x, lane, s_result[warpid+k*8]);
27   }
28   __syncthreads();
29
30   // Reducción de las reducciones.
31   if(warpid==0) {
32     tb_reduceWarp(s_data, thid, lane, s_data[0]);
33   }
34   // La reducción del bloque se queda guardada en s_data[0] .
35 }
```

Figura 2.7: Algoritmo de reducción del kernel de TB4-warp.

scan de Dotsenko et al. [DGS+08]. En este algoritmo, cada hilo realiza secuencialmente la reducción de una lista consecutiva de datos (línea 16) cuyo comienzo ha sido previamente localizado (línea 11). Si el bloque de datos se viera como una matriz de tamaño $W \times H$ (guardada por filas) entonces cada hilo se encargaría de reducir una fila. Debido a este procesamiento del bloque de datos, este algoritmo recibe el nombre de *Matrix*. Al igual que en TB4-warp, los resultados parciales de cada fila se vuelven a reducir, utilizando nuevamente `tb_reduceWarp` (línea 23).

Se ha establecido $W = 32$ y $H = 32$ para el tamaño de la matriz porque supone un buen compromiso entre el número de warps que reducen y el número de elementos que reduce cada hilo. Con esta configuración, solo un warp se encarga de reducir un bloque de datos, cuyo tamaño es $D = W \times H = 1024$. Al igual que en TB2, se podría producir conflictos de banco durante el recorrido de cada fila, pero esto se soluciona añadiendo un padding de un elemento por fila (línea 10).

El cálculo de direcciones es más simple en Matrix que en TB2 ó TB4-warp debido a que el procesamiento de los datos se realiza secuencialmente por fila. Además, ya que solo un warp se encarga de reducir todas las filas de la matriz, no es necesaria ninguna sincronización explícita. Sin embargo, Matrix tiene el inconveniente de que solo un warp está activo durante la mayor parte del proceso, lo que dificulta el aprovechamiento del multithreading.

```
void Matrix_reduction (float* s_data){
  // ID del hilo.
  unsigned int thid = threadIdx.x;
  // Acumulador de la reducción.
  float current_red;

  // Solo el primer warp reduce.
  if(thid < H){
    // Comienzo de la fila.
    unsigned int base = thid * (W + PADDING);
    float* row = &s_data[base];
    // Primer valor de la fila.
    current_red = row[0];

    // Reducción secuencial.
    for(unsigned int k = 1; k < W; k++) {
      current_red = op(current_red, row[k]);
    }
    // Guardamos la reducción parcial.
    s_data[thid] = current_red;

    // Reducción intra-warp.
    tb_reduceWarp(s_data, thid, thid&31, s_data[0]);
  }
  // La reducción del bloque se queda guardado en s_data[0].
}
```

Figura 2.8: Algoritmo de reducción del kernel de Matrix.

2.5.2. Reducción Segmentada

La *reducción segmentada* es una primitiva que realiza la reducción de los datos de cada *segmento* de la entrada. Para ello, asumimos que el array de entrada está dividido en segmentos cuyos elementos se encuentran en posiciones consecutivas. La salida de esta primitiva es un array con tantos elementos como segmentos tiene la entrada. Asumimos también que el array de salida, `g_output`, se encuentra en memoria global y que sus componentes están inicializadas con el elemento neutro $1_{\oplus}$.

En esta sección, los segmentos se han implementado añadiendo un nuevo array de enteros, llamado *owners*. Owners tiene tantos elementos como el array de entrada, lo que permite establecer una correspondencia uno-a-uno entre los dos arrays. Cada elemento de owners indica el índice del segmento al que pertenece el elemento de entrada (un ejemplo se muestra en la figura 2.9). Una implementación de los segmentos alternativa a los owners se realiza mediante *headflags*. El array headflags es, al igual que owners, un array de enteros del mismo tamaño que la entrada. En este caso, el headflag es 1 si se trata del comienzo del segmento y 0 en otro caso. A partir de headflags se puede obtener owners si se le aplica un scan inclusivo y se resta uno.

Reducción segmentada tree-based

Al igual que con la reducción no segmentada, nos basamos en una ecuación recursiva para implementar su versión segmentada, en cuyas hojas se encuentra la información de entrada (fun-

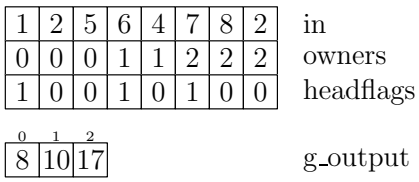


Figura 2.9: Ejemplo de un array de entrada segmentado. Los segmentos se han implementado con un array adicional mediante `owners` y `headflags`. La salida de este array se encuentra en `g_output`.

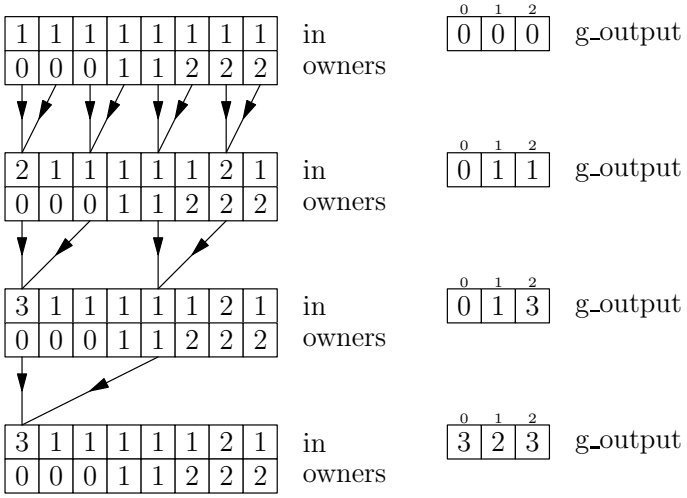


Figura 2.10: Ejemplo de reducción segmentada de un bloque con $B = 4$ y $D = 8$. La operación usada como $\oplus$ es la suma. El último hilo activo se encarga de acumular el valor de `in[0]` en `g_output[0]`. El resto de valores se han acumulado durante la ejecución con la operación †.

ción *value*) y de owners (función *owner*):

$$s\_red(n) = \begin{cases} s\_red(n.l) \oplus s\_red(n.r) & \text{si } n \text{ es interno y } owner\_rec(n.l) = owner\_rec(n.r) \\ s\_red(n.l)^{\dagger} & \text{si } n \text{ es interno y } owner\_rec(n.l) \neq owner\_rec(n.r) \\ value(n) & \text{si } n \text{ es hoja} \end{cases}$$

$$owner\_rec(n) = \begin{cases} owner\_rec(n.l) & \text{si } n \text{ es nodo interno} \\ owner(n) & \text{si } n \text{ es hoja} \end{cases} \tag{2.2}$$

Las funciones *s_red* y *owner_rec* establecen los valores de reducción y de owner de cada nodo del árbol. Además, como señalan Zhou et al. [ZHWG08], es necesario realizar una operación extra cuando un hilo intenta reducir dos valores provenientes de segmentos diferentes. Esta operación adicional está marcada con el símbolo † en la ecuación 2.2 y consiste en actualizar la salida acumulando el valor de reducción del hijo derecho, es decir, realizando la asignación siguiente

$$\dagger \equiv g\_output[owner\_rec(n.r)] = g\_output[owner\_rec(n.r)] \oplus s\_red(n.r)$$

Un ejemplo sencillo se puede ver en la figura 2.10.

No es necesario utilizar operaciones atómicas para implementar la operación † ya que se tiene garantía de que dos o más hilos no van a acceder a la misma posición en `g_output`. La demostración

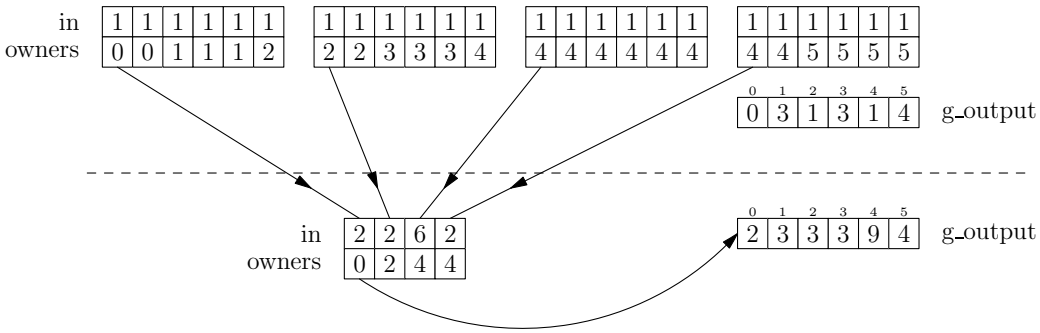


Figura 2.11: Ejemplo de reducción segmentada recursiva con $D = 6$. Dos pasadas son necesarias para reducir la entrada de 24 elementos. Al final de la primera pasada se guarda en memoria global la reducción del primer segmento de cada bloque de datos. Al final de la segunda pasada, la reducción del primer segmento del único bloque de datos se guarda directamente en g_output[0].

se puede hacer por contradicción, sabiendo que el array de owners se encuentra siempre ordenado. Supongamos que el hilo i reduce dos elementos cuyos owners son a y b, tales que $a < b$. Por la operación †, ese hilo actualizará la componente b de `g_output`. Si otro hilo j, $j \neq i$, intentara modificar ese valor es porque querría reducir dos elementos con owners c y b, tales que $c < b$. Ya que la lista de owners se encuentra siempre ordenada, solo pueden ocurrir dos casos al comparar los índices de los dos hilos: si $i < j$ entonces $(a < b \leq c < b)$; si $i > j$ entonces $(c < b \leq a < b)$. En cualquiera de los casos se llega a la contradicción $b < b$, concluyendo que nunca dos hilos diferentes querrán actualizar la misma componente de `g_output`.

El elemento resultante de la reducción del segmento que se encuentra más a la izquierda en el array en memoria compartida (es decir, la reducción del primer segmento del bloque de datos) nunca se guarda en `g_output` por medio de la operación † de la ecuación 2.2. Así, el último hilo activo es el encargado de guardarlo en memoria global. Si un único bloque de hilos es suficiente para reducir toda la entrada, entonces ese hilo acumula el valor directamente en `g_output[0]`. Si son necesarios varios bloques, entonces ese valor junto con su owner se guarda en un array intermedio, que será posteriormente reducido. En una de esas reducciones segmentadas posteriores, ese valor se acumulará en `g_output` mediante la operación † o por el último hilo activo del último bloque (ver ejemplo en la figura 2.11).

En Martín et al. [MATG12] hemos adaptado los algoritmos tree-based TB2 y TB4-warp a su versión segmentada. Esta adaptación es sencilla: solo hay que realizar la operación † siempre que se reduzcan dos elementos con diferentes owners. Así, hay que sustituir la línea 22 de TB2 (figura 2.5) por el código de la figura 2.12. En TB4-warp, ya que el código está desplegado, hay que sustituir las líneas 5, 7, 9, 11 y 13 del algoritmo de la figura 2.7 por una llamada al código de la figura 2.13.

Reducción segmentada secuencial

Adaptar el algoritmo Matrix para que procese segmentos es también sencillo. solo hemos tenido que sustituir el código de las líneas 11–23 del algoritmo de la figura 2.8 por el código que se muestra en la figura 2.14. En este caso, cada hilo tiene en cuenta el owner del último elemento procesado (línea 6) durante el recorrido de una fila. Si durante este recorrido se produce un cambio de segmento (línea 10), entonces el último valor de reducción se acumula en `g_output` (línea 12), lo que equivale a la operación † de la ecuación 2.2. El recorrido de la fila continúa, pero comenzando

```
1  unsigned int lo = s_owner[ai]; // owner izquierda
2  unsigned int ro = s_owner[bi]; // owner derecha
3  // Comprobamos si los owners son iguales.
4  if(lo != ro) {
5    // Actualización de la salida.
6    g_output[ro] = op(g_output[ro], s_data[bi]);
7  } else {
8    // Reducción de dos elementos del mismo segmento.
9    s_data[ai] = op(s_data[ai], s_data[bi]);
10 }
```

Figura 2.12: Código que hay que añadir a TB2 para que realice la reducción segmentada.

```
1  void sreduce_leftstoring(float* g_output,
2        unsigned int oLeft,  float &dLeft,
3        unsigned int oRight, float dRight){
4    // Comprobamos si los owners son iguales.
5    if(oLeft != oRight) {
6      // Actualizacion de la salida.
7      g_output[oRight] = op(g_output[oRight], dRight);
8    } else {
9      // Reducción de dos elementos del mismo segmento.
10     dLeft = op(dLeft, dRight);
11   }
12 }
```

Figura 2.13: Código que hay que añadir a TB4-warp para que realice la reducción segmentada.

la reducción del siguiente segmento (líneas 14-15).

Al igual que en Matrix, se tiene que realizar una reducción segmentada de los últimos valores de reducción de cada hilo, es decir, de la reducción del segmento de más a la derecha en cada fila. Para llevar a cabo esta tarea, hemos usado también la reducción segmentada `tb_sreduce_rightstoring` (línea 25), aunque, debido a que el procesamiento de las filas se realiza de izquierda a derecha, esta reducción debe ser right-storing. Además, el valor de salida de cada bloque de hilos de la pasada actual es el elemento de más a la derecha que se obtiene tras haber aplicado la operación `tb_sreduce_rightstoring`.

2.5.3. Optimizaciones Algorítmicas

En nuestro artículo de Martín et al. [MATG12] presentamos dos optimizaciones algorítmicas que se pueden integrar en cualquiera de los algoritmos de reducción anteriores. Sus nombre son *bloques persistentes* y *productor-consumidor*.

Bloques Persistentes. El objetivo detrás de esta optimización consiste en reducir el número de elementos que se generan en cada pasada. De esta forma, se necesitan menos pasadas para reducir la entrada completa, disminuyendo, por tanto, el sobrecoste del tráfico de datos entre el chip y la memoria global. Una manera de implementar esta optimización consiste en que cada bloque de hilos se encargue de reducir más de un bloque de datos. Así, a diferencia de los algoritmos anteriores, cada bloque reduce secuencialmente varios bloques de datos, acumulando los resultados de cada uno en memoria compartida.

Para implementar este procesamiento secuencial hemos seguido el esquema de los hilos persistentes[9]. Se trata de lanzar a ejecución el máximo número de bloques de hilos de manera que

[9]Este algoritmo está basado en el trabajo de Aila y Laine [AL09] y sus detalles se describirán en la sección 4.3.1

```
// Comienzo de la fila de los datos y los owners.
float*        dRow = &s_data[base];
unsigned int* oRow = &s_owner[base];
// Iniciamos la reducción con el primer elemento de la fila.
current_red   = dRow[0];
current_owner = oRow[0];
// Recorrido secuencial de cada fila.
for(unsigned int k = 1; k < W; k++){
  // Comprobamos el owner actual con el siguiente.
  if(current_owner != oRow[k]){
    // Si los owners son diferentes, acumulamos el valor en memoria global.
    g_output[current_owner] = op(current_red, g_output[current_owner]);
    // Comenzamos un nuevo segmento.
    current_red   = dRow[k];
    current_owner = oRow[k];
  } else {
    // Si los valores de los owners son iguales, seguimos acumulando .
    current_red   = op(current_red, dRow[k]);
  }
}
// Almacenamiento de la ultima reduccion de la fila
s_data[thid] = current_red;
s_owner[thid] = current_owner;
// Reducimos los ultimos segmentos de cada fila
tb_sreduce_rightstoring(s_data, s_owner, g_output,
        thid, thid&31, s_data[0], s_owner[0]);
```

Figura 2.14: Modificación del código de Matrix para que realice la reducción segmentada.

ninguno de ellos se quede esperando debido a una escasez de recursos del chip (registros y memoria compartida). La asignación de cada bloque de datos al bloque de hilos que lo reducirá se realiza de manera estática, ya que todos los bloques de datos requieren el mismo procesamiento.

Aunque, en general, se puede modificar el número de bloques de hilos lanzados y, en consecuencia, el número de bloques de datos asignados a cada uno, los parámetros requeridos por los hilos persistentes son un buen compromiso entre el aprovechamiento de los recursos y el número de elementos de salida. Así, disminuir el número de hilos por bloque implica lanzar más bloques de hilos y supone que algunos se queden esperando y que se produzcan más elementos de salida para la siguiente pasada. Lanzar más hilos por bloque requiere menos bloques y, por tanto, se obtienen menos elementos de salida, aunque se dispondría de menos warps para aprovechar el multithreading[10].

Productor-Consumidor. Esta técnica, que se realiza sobre la optimización anterior de los bloques persistentes, permite solapar la lectura del siguiente bloque de datos con la reducción del bloque actual. En el esquema de los bloques persistentes, todos los hilos se encargan primero de traer los datos a memoria compartida para ser posteriormente reducidos. Por tanto, es necesario garantizar, mediante una barrera, que los datos estén disponibles antes de la reducción, secuencializándose las etapas de lectura de memoria y de reducción.

En la optimización productor-consumidor, algunos warps se encargan de reducir un bloque de datos (*warps consumidores*) mientras que otros se encargan de traer el siguiente bloque de datos desde memoria (*warps productores*). Warps de ambos tipos pueden intercalar su ejecución libremente ya que no trabajan sobre los mismos datos. De esta forma, es posible esconder la latencia de memoria solapando la reducción de un bloque de datos con la lectura del siguiente.

[10]En un caso más extremo, el número de bloques de hilos lanzados a ejecución es tan bajo que algunos multiprocesadores no se utilizan, lo que implica desaprovechar la potencia de cálculo del hardware.

```
1  // Cargamos el primer bloque de datos.
2  if(warpid >= C)
3    loadChunk(first, s_load);
4  __syncthreads();
5  // Indice del siguiente bloque de datos.
6  first += D;
7
8  // Bucle productor-consumidor.
9  for(unsigned int k = 1; k < d; k++) {
10   // Intercambia los buffers de memoria compartida.
11   aux = s_load;  s_load = s_comp;  s_comp = aux;
12   // Cada warp consume o produce.
13   if(warpid < C)
14     reduceChunk(s_comp, s_current_red);
15   else
16     loadChunk(first, s_load);
17   __syncthreads();
18   first += D;
19 }
20
21 // Reduce el ultimo bloque de datos.
22 if(warpid < C)
23   reduceChunk(s_load, s_current_red);
24 __syncthreads();
25 // El resultado final se encuentra en s_current_red.
```

Figura 2.15: Esquema productor-consumidor. La función `reduceChunk` ejecuta cualquiera de los algoritmos de reducción anteriores, mientras que `loadChunk` se encarga de leer el siguiente bloque de datos.

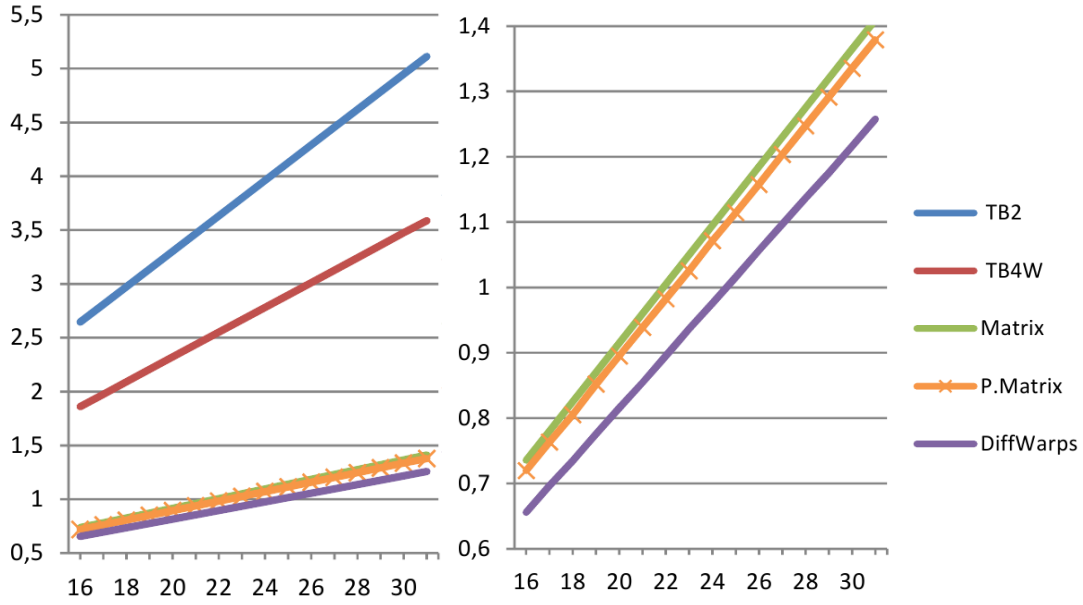
El código de este algoritmo se muestra en la figura 2.15. La constante C (líneas 2, 13 y 22) indica que los primeros C warps son los consumidores, mientras que el resto son los productores. Los warps consumidores se encargan de reducir el bloque de datos actual (línea 14) mientras que los productores traen datos desde memoria (línea 16). Son necesarios dos buffers en memoria compartida para implementar este esquema: uno mantiene el bloque de datos actual que se reduce, `s_comp`, y el otro guarda los datos del siguiente bloque, `s_load`. Estos buffers se intercambian cuando los productores y consumidores han terminado sus tareas (línea 11). El primer y el último bloque de datos se tienen que procesar independientemente (líneas 2-4 y 22-24).

2.5.4. Resultados

Hemos probado experimentalmente los algoritmos de reducción presentados en las secciones anteriores. Los resultados se pueden encontrar en nuestro trabajo de Martín et al. [MATG12]. El hardware usado para las pruebas es una NVidia GTX 480 (Fermi, cap. 2.0) con 1536MB de RAM. Los 64KB de la memoria interna de cada multiprocesador se han configurado para que 48KB se usen para memoria compartida y 16KB para caché L1. Los datos que se muestran están obtenidos con la configuración de caché por defecto, es decir, las cachés L1 y L2 están habilitadas. Se han conseguido resultados similares deshabilitando la caché L1, aunque no se muestran.

Reducción no segmentada

Se han probado los tres algoritmos de reducción no segmentada TB2, TB4-warp y Matrix. Las optimizaciones de bloques persistentes y productor-consumidor solo se han implementado sobre Matrix, ya que este es el algoritmo de reducción no segmentado más rápido. Los nombres de estos



(a) (b)

Figura 2.16: Fig. (a). Resultados para los cinco algoritmos de reducción no segmentada. En el eje x se muestra el número de elementos de la entrada, donde la unidad representa un tamaño de 2^{20} elementos. En el eje y se muestra el tiempo total de los algoritmos, medido en *ms*. Fig. (b). Resultados de los algoritmos Matrix, P_Matrix y Diffwarps únicamente.

algoritmos son *P_Matrix* si usa la optimización de los bloques persistentes, y *Diffwarps* si usa los bloques persistentes más la técnica del productor-consumidor.

Estos algoritmos se han probado sobre un array de `floats` de tamaño $n = m * 2^{20}$, donde m varía entre 16 y 31. El número de pasadas que requieren TB2, TB4-warp y Matrix es siempre de 3, mientras que P_Matrix y Diffwarps requieren solo 2 porque cada bloque de hilos consume varios bloques de datos. Se han probado dos operaciones para $\oplus$: el mínimo y la suma de `floats`. Ambas tienen un rendimiento similar, por lo que solo se mostrarán los datos de la operación mínimo.

En la figura 2.16 se muestran los resultados experimentales de todos estos algoritmos. Estos resultados muestran que TB4-warp es más rápido que TB2 y que Matrix es mucho más rápido que cualquiera de las dos reducciones tree-based. Las conclusiones que obtenemos es que el uso de barreras produce un gran impacto en el rendimiento. Así, TB4-warp ejecuta muchas menos barreras que TB2, mientras que Matrix no realiza ninguna. Por tanto, Matrix obtiene las mejores ganancias ya que es el algoritmo cuyo cálculo de direcciones es el más sencillo.

Cualquiera de las dos optimizaciones algorítmicas presentadas permite bajar el número de pasadas necesarias para reducir la entrada desde 3 hasta 2. Esta disminución consigue, por tanto, reducir el tiempo que tarda la reducción completa al evitar la sobrecarga del tráfico con memoria entre cada pasada. Diffwarps, al ser una optimización que se realiza sobre P_Matrix, permite disminuir aún más su tiempo de ejecución debido al solapamiento entre la reducción de un bloque de datos y la lectura del siguiente.

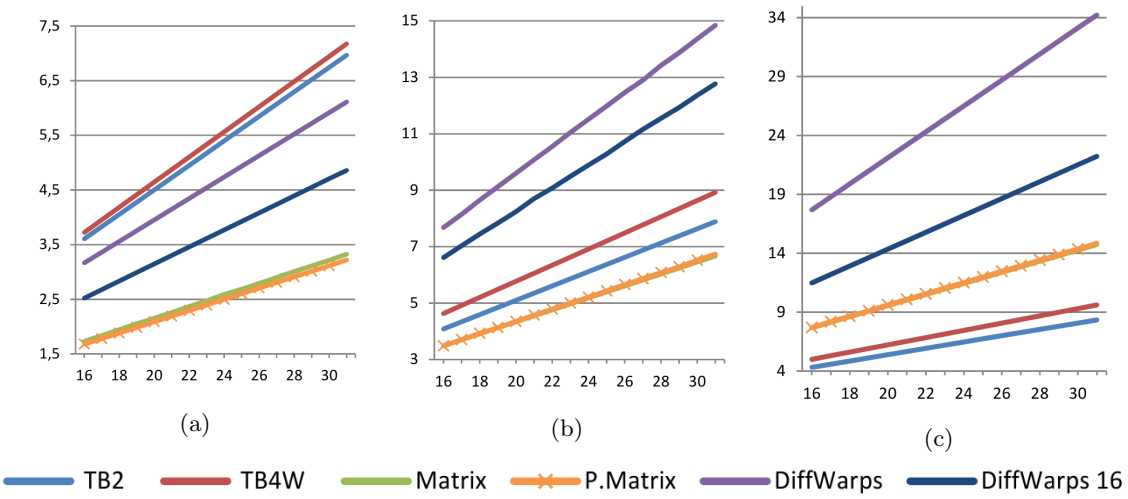


Figura 2.17: Resultados para los cinco algoritmos de reducción segmentada. En el eje x se muestra el número de elementos de la entrada, donde la unidad representa un tamaño de 2^{20} elementos. En el eje y se muestra el tiempo total de los algoritmos, medido en *ms*. Fig. (a). Resultados cuando la entrada consta de un único segmento. Fig. (b). Resultados cuando los segmentos varían aleatoriamente entre 10 y 50 elementos. Fig. (c). Resultados cuando todos los segmentos están formados por 3 elementos.

Reducción segmentada

Hemos probado los cinco algoritmos anteriores en sus contrapartidas segmentadas. Estas versiones requieren el doble de espacio de memoria compartida que sus versiones no segmentadas debido a que, por cada dato, hay que mantener también su owner. Este aumento en la demanda de memoria compartida supone que no es posible alcanzar la ocupación máxima (ocupación del 100 %) en la mayoría de los algoritmos. El caso peor se encuentra en Diffwarps segmentado, cuya ocupación cae hasta el 38 %, ya que tiene que reservar espacio para dos bloques de datos más los owners de cada dato. Por tanto, hemos desarrollado una versión con más ocupación disminuyendo el número de datos que procesa cada hilo secuencialmente. En concreto, W disminuye desde 32 a 16. A este nuevo algoritmo le hemos llamado $Diffwarps_{16}$, y tiene una ocupación del 94 %.

El rendimiento de los algoritmos de reducción segmentada se ve afectado por la distribución de los segmentos sobre la entrada. Así, cuantos más pequeños sean estos segmentos más veces se va a actualizar `g_output` y, por tanto, más lecturas y escrituras a memoria principal se producen. Vamos a analizar tres escenarios. En el escenario (a) solo existe un segmento, por lo que no se producirán actualizaciones a `g_output`. En el escenario (b), se generan aleatoriamente segmentos de tamaño entre 10 y 50. En el escenario (c), los segmentos son siempre de tamaño 3, lo que produce muchos accesos a memoria global.

Los resultados de los tres escenarios se muestran en la figura 2.17. En el escenario (a) no se producen accesos a memoria global durante la reducción ya que no existen datos con owners diferentes. Por tanto, es de esperar que el rendimiento sea similar al caso no segmentado. Esto sucede en general, aunque hay dos excepciones. Por un lado Diffwarps ya no es el más rápido debido a que su ocupación ha caído. Su versión Diffwarps_{16} tiene el inconveniente de que realiza la mitad del trabajo que hacía Diffwarps con los mismos hilos, por lo que tampoco es competitivo. Por otro lado, TB2 es mejor que TB4-warp. TB4-warp presenta más divergencias dentro de los warps

que TB2, aunque con menos sincronizaciones. Estas divergencias son más costosas en la versión segmentada que en la no segmentada, ya que se ejecuta más código por hilo, lo que es suficiente para que el rendimiento de TB4-warp esté por debajo del de TB2.

A medida que los segmentos se hacen más pequeños (escenarios (b) y (c)) se producen más operaciones †, por lo que todos los algoritmos se vuelven más lentos. Sin embargo, los tree-based no se ven tan perjudicados como los secuenciales. Incluso, en el escenario (c), los tree-based obtienen un rendimiento mejor que el resto. Esto se debe a que existen más oportunidades de que se produzcan escrituras fusionadas a memoria durante la actualización del array `g_output` en los algoritmos tree-based que en los secuenciales. Así, en las primeras etapas de los tree-based, muchos hilos se encargan de reducir elementos muy próximos en el array de datos. Si estos elementos pertenecen a segmentos diferentes —lo que es muy probable en el escenario (c)— entonces las escrituras a `g_output` se fusionarán. Sin embargo, en los secuenciales cada hilo de un warp reduce elementos que se encuentran a una distancia de W. En el escenario (c), debido a que $W = 32$ y los segmentos son de 3 elementos, solo los hilos que se encuentran a una distancia de tres filas escriben en `g_output`. Por tanto, las direcciones que se escriben en `g_output` se encuentran suficientemente lejos como para que se lance una transacción por acceso sin posibilidad de fusión.

2.6. Algoritmo de Dijkstra Paralelo

Un problema clásico en teoría de grafos es el problema SSSP (*Single-Source Shortest Paths*). Dado un grafo dirigido y valorado, este problema consiste en encontrar el camino con menor peso[11] desde un único nodo, llamado nodo *fuente*, hasta todos los demás. El algoritmo de Dijkstra resuelve este problema cuando los pesos de las aristas son positivos. Para simplificar el desarrollo de esta sección, vamos a suponer también que los pesos son estrictamente mayores que cero y que el nodo fuente siempre posee el índice 0.

2.6.1. Algoritmo de Dijkstra Clásico o de Frontera Simple

En el algoritmo de Dijkstra, a cada nodo se le asocia un valor que indica la *estimación* del camino con menor peso desde el nodo fuente hasta él mismo. Vamos a suponer que existe un valor de estimación mayor que cualquier otro, al que llamamos `INFINITY`. Si no se ha encontrado todavía ningún camino desde la fuente hasta el nodo, entonces a ese nodo le corresponde una estimación de `INFINITY`.

Este algoritmo clasifica los nodos en dos conjuntos disjuntos: los *resueltos* y los *no resueltos*. Los nodos resueltos son aquellos para los que ya se conoce el peso de su camino más corto. Los nodos no resueltos no conocen todavía su camino más corto pero sí una estimación del peso de ese camino. Esa estimación para los no resueltos siempre es una cota superior del peso de su camino más corto.

En cada iteración del algoritmo, existe un nodo destacado que recibe el nombre de nodo *frontera*. Ese nodo es elegido de entre los no resueltos como el nodo con menor estimación. En caso de que existan varios nodos con la misma estimación mínima, se elige uno cualquiera. Después de esta elección, se intenta disminuir la estimación de todos los nodos sin resolver que son accesibles (*sucesores*) desde la frontera. Esta operación recibe el nombre de *relajación* y el acto de intentar disminuir la estimación de un nodo se llama *relajar*. Para implementar la relajación, se comprueba si la estimación del nodo frontera más el peso de la arista que lo une a un nodo sucesor no resuelto es menor que la estimación ya guardada en el nodo sucesor. Después, el nodo frontera se añade a

[11] También llamaremos *camino más corto* al camino con menor peso, independientemente del número de arcos que posea.

```
1 void Dijkstra() {
2     init(c,f,u);
3     mssp = 0;
4     while(mssp != INFINITY) {
5         relax(c,f,u);
6         mssp = minimum(c,u);
7         update(c,f,u,mssp);
8     }
9 }
```

Figura 2.18: Esquema general del algoritmo de Dijkstra.

los resueltos ya que se tiene garantía de que su estimación no va a disminuir más durante el resto del algoritmo.

El algoritmo de Dijkstra sigue el esquema de la figura 2.18. El algoritmo mantiene dos arrays del tamaño del número de nodos del grafo: `c` y `u`. Cada nodo está especificado como un entero que, además, sirve como índice sobre los dos arrays anteriores. Así, `c[i]` es un entero que indica la estimación del camino más corto desde el nodo fuente hasta el nodo `i`. El nodo `i` es un nodo no resuelto si su valor booleano `u[i]` es cierto, y resuelto en caso contrario. El nodo frontera es aquel cuyo índice coincida con el valor `f`. Por simplicidad, suponemos que el índice del nodo fuente es siempre 0.

La función `init` (línea 2) se encarga de inicializar los arrays `c` y `u` junto con el nodo frontera `f`. Al comienzo del algoritmo todos los nodos están no resueltos y tienen estimación `INFINITY`, con excepción del nodo fuente que se considera resuelto y con estimación 0. El nodo frontera `f` es inicializado al nodo fuente, es decir, con el valor 0. La función `relax` (línea 5) intenta relajar la estimación de todos los nodos sucesores no resueltos `j` de la frontera `f` mediante la operación `c[j] = min(c[j],c[f]+w[f,j])`, donde `w[i,j]` es el peso de la arista que une el nodo `i` con el nodo `j`, en caso de que la hubiera. La función `minimum` (línea 6) encuentra la estimación mínima, `mssp`, entre todos los nodos no resueltos. La función `update` (línea 7) actualiza el nodo frontera escogiendo un nodo cualquiera con estimación `mssp` de entre los no resueltos. Esta función también se encarga de añadir el nodo frontera a los resueltos. Por último, el camino más corto para todos los nodos ya se ha encontrado si `mssp` es `INFINITY` (línea 4).

2.6.2. Algoritmo de Dijkstra Paralelo o de Frontera Compuesta

Se puede comprobar en el algoritmo de Dijkstra que se cumple el siguiente aserto: todos los nodos no resueltos que tienen la misma estimación que la frontera van a ser elegidos posteriormente como nuevos nodos frontera antes que cualquier otro nodo (ver Martín et al. [MTG09] y Martín et al. [MTG08] para más detalles). Eso se ve fácilmente comprobando que la estimación de estos nodos no puede disminuir más[12]. De esta forma, el conjunto de todos los nodos no resueltos que tienen la misma estimación que `mssp` pueden gestionarse simultáneamente, en vez de considerarlos independientemente. A este conjunto de nodos lo vamos a llamar *frontera compuesta*.

La gestión de la frontera compuesta requiere sustituir el índice `f` por un array de booleanos cuyo tamaño es el número de nodos del grafo. Ahora se puede determinar si el nodo `i` pertenece a la frontera consultando su valor booleano `f[i]`. Además, el concepto de frontera compuesta sirve para paralelizar la función `relax`. Primero veamos un par de algoritmos secuenciales que implementan `relax`.

El primer algoritmo, `relax_suc` (figura 2.19), recorre todos los nodos frontera y relaja la

[12] Una forma de ver esto es suponer que la estimación de esos nodos va a ser relajada por otro nodo no resuelto. Sin embargo, eso no es posible ya que el camino es más largo y los pesos de las aristas son mayores que cero.

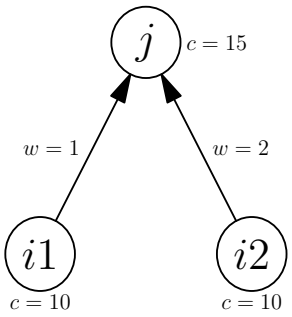
```
1 void relax_suc(c, f, u) {
2     forall nodo i
3         if(f[i])
4             forall j sucesor de i do
5                 if(u[j])
6                     c[j] = min(c[j], c[i] + w[i,j]);
7 }
```

```
1 void relax_pre(c, f, u) {
2     forall nodo j
3         if(u[j])
4             forall i predecesor de j do
5                 if(f[i])
6                     c[j] = min(c[j], c[i] + w[i,j]);
7 }
```

Figura 2.19: Dos algoritmos secuenciales que implementan la función `relax`: `relax_suc` recorre los nodos frontera y relaja sus sucesores; `relax_pre` recorre los nodos no resueltos y se relajan si son sucesores de algún nodo frontera.

estimación de sus sucesores. El segundo algoritmo, `relax_pre` (figura 2.19), recorre todos los nodos no resueltos y analiza si son sucesores de algún nodo frontera, relajando entonces su propia estimación. La posibilidad de acceder a los sucesores o predecesores de un nodo depende de la implementación del grafo. Si el grafo está implementado como una lista de adyacencia, entonces cada nodo solo puede conocer o sus sucesores o sus predecesores, según el caso. Si el grafo está implementado como una matriz de adyacencia, entonces cada nodo puede conocer tanto sus predecesores como sus sucesores, simplemente recorriendo una fila o una columna de la matriz.

El bucle de la línea 2, tanto en `relax_suc` como en `relax_pre`, puede implementarse secuencialmente, simplemente recorriendo todos los nodos del grafo. Por otro lado, esos bucles también pueden realizarse en paralelo simplemente lanzando un hilo por cada nodo del grafo que se procesa en el cuerpo del bucle de la línea 4. Sin embargo, existe el problema de que cada nodo no es completamente independiente de los demás, debido a la dependencia que implica las líneas 6. En la linea 6 de `relax_pre` se puede dar el caso de que dos nodos `j` consulten los valores `c[i]` y `w[i,j]` para un mismo nodo frontera `i`. Sin embargo, eso no es ningún problema ya que dos lecturas pueden realizarse en paralelo sin riesgo. En la línea 6 de `relax_suc` se puede dar el caso de que dos nodos frontera `i` intenten relajar el mismo nodo `j` no resuelto. Si eso es así puede haber inconsistencias debido a la concurrencia. Supongamos el siguiente ejemplo para `relax_suc` en el que dos nodos frontera `i1` y `i2` intentan relajar el mismo nodo no resuelto `j`




```
Estado de partida:
  c[i1] = 10, c[i2] = 10, w[i1,j] = 1, w[i2,j] = 2, c[j] = 15
```

```
1  void init(c, f, u) {
2      forall nodo i {
3          if(i == 0) {
4              c[i] = 0;
5              f[0] = true;
6              u[0] = false;
7          } else {
8              c[i] = INFINITY;
9              f[i] = false;
10             u[i] = true;
11         }
12     }
13 }
```

```
1  void update(c, f, u, mssp) {
2      forall nodo i {
3          f[i] = false;
4          if (c[i] == mssp){
5              u[i] = false;
6              f[i] = true;
7          }
8      }
9  }
```

Figura 2.20: Procedimientos `init` y `update` del algoritmo paralelo de Dijkstra.

```
Posible secuencia de instrucciones:
  1 - nodo i1: lectura de c[j], se guarda en reg1 (reg1=15)
  2 - nodo i2: lectura de c[j], se guarda en reg1 (reg1=15)
  3 - nodo i1: lectura de c[i1] y w[i1,j], su suma se guarda en reg2 (reg2=11)
  4 - nodo i2: lectura de c[i2] y w[i2,j], su suma se guarda en reg2 (reg2=12)
  5 - nodo i1: operación min entre reg1 y reg2, se guarda en reg3 (reg3=11)
  6 - nodo i2: operación min entre reg1 y reg2, se guarda en reg3 (reg3=12)
  7 - nodo i1: guarda reg3 en c[j] (c[j]=11)
  8 - nodo i2: guarda reg3 en c[j] (c[j]=12)

Resultado:
  c[j] = 12   ¡¡en vez de 11!!
```

Este problema, típico en los sistemas paralelos y concurrentes, se conoce como riesgo de lectura después de escritura. El problema se debe fundamentalmente a que asignar a una variable el mínimo de otras dos no es una operación atómica, es decir, está formada por varias instrucciones y cada una de ellas se puede intercalar con las de otros hilos de cualquier manera. Obsérvese que la lectura del paso 2 se debería haber ejecutado después de que el hilo asociado al nodo `i1` hubiera guardado su valor en `c[j]` en el paso 7. En consecuencia, la solución consiste en que la ejecución de esa instrucción sea atómica, es decir, se realice sin interferencias por parte de otros hilos. Las GPUs con cap. 1.0 no poseen tales instrucciones atómicas, pero afortunadamente las GPUs a partir de cap. 1.1 sí poseen una operación de mínimo atómica, llamada `atomicMin`. De esta forma, el algoritmo `relax_suc` debe ser implementado con esta operación para garantizar su corrección. Por el contrario, el algoritmo `relax_pre` no es necesario que la use, ya que no existe la posibilidad de que ocurra un riesgo de lectura después de escritura. Este análisis es importante ya que puede dar lugar a implementaciones erróneas, como se puede ver en Harish y Narayanan [HN07].

Por otra parte, los procedimientos `init` y `update` son fácilmente paralelizables, ya que los valores `c`, `f` y `u` de cada nodo pueden actualizarse en paralelo (figura 2.20). El procedimiento `minimum` consiste en obtener el mínimo de las estimaciones de todos los nodos no resueltos. La primitiva paralela *reducción* (sección 2.5) se puede adaptar para que implemente la operación `minimum`. Es suficiente con aplicar la reducción sobre el array de estimaciones `c`, usando *min* (mínimo de dos enteros) como operación de reducción. Sin embargo, solo se desea calcular el mínimo de las estimaciones para los nodos no resueltos. Esto se consigue fácilmente si, en el momento de la lectura de la estimación de un nodo también se comprueba si está resuelto. En caso de que lo esté,

el valor cargado desde memoria será el elemento neutro de *min*, es decir, `INFINITY`. Esto permite que la estimación de los nodos resueltos no afecte a la evaluación de `mssp`.

2.6.3. Resultados Experimentales

Se han implementado versiones en CPU y en GPU de los algoritmos de Dijkstra antes descritos. Los algoritmos en CPU se han probado sobre un procesador Intel Core Quad Q6600 de 2.40GHz con 2GB de RAM. Los algoritmos en GPU usan el procesador anterior junto con una NVidia GeForce GTX 280 con 1GB de RAM. Los algoritmos que implementan el procedimiento `relax_pre` para relajar los nodos no resueltos reciben el nombre de U, mientras que los algoritmos que usan `relax_suc` se denotan con F. Los algoritmos U^{CPU} y F^{CPU} implementan enteramente en CPU los procedimientos `minimum`, `relax` y `update` para fronteras compuestas. Además, también se ha implementado en CPU una versión clásica (frontera simple) del algoritmo de Dijkstra que usa un *Fibonacci Heap* para organizar los nodos del grafo en función de su estimación. Este algoritmo recibe el nombre de FH. Un Fibonacci Heap es una estructura que permite una implementación eficiente en CPU debido a su coste amortizado del orden de $O(\log(n))$ para recuperar la estimación mínima, y $\Theta(1)$ para relajar un nodo.

Se han probado también las versiones con frontera compuesta implementadas en GPU mediante CUDA. Estos algoritmos reciben el nombre de U^{GPU}, si usa `relax_pre`, y F^{GPU}, si usa `relax_suc`. El procedimiento `minimum` requiere varias pasadas para obtener la estimación mínima de los nodos no resueltos. La primera pasada se realiza en GPU y su salida se envía a la CPU para obtener el valor de minimización final. En caso de que este valor sea `INFINITY`, se saldrá del bucle y el algoritmo habrá terminado. Además, se ha implementado una versión de F^{GPU}, denotada como $F^{GPU\_no\_Atomic}$, en la que la operación mínimo de `relax` no es atómica. Como se ha comentado antes, esta implementación es incorrecta, aunque su ejecución muestra la penalización debida al uso de operaciones atómicas.

Los grafos se han implementado mediante una matriz o una lista de adyacencia. En el caso de la lista de adyacencia, cada nodo mantiene una lista que guarda sus nodos sucesores o predecesores, según el caso. El grafo tiene que estar implementado como una lista de sucesores adyacentes si los algoritmos usan `relax_suc` para relajar. Si usan `relax_pre`, entonces el grafo debe estar implementado mediante una lista de predecesores adyacentes. En el caso de la matriz de adyacencia, se pueden usar cualquiera de las dos versiones de `relax` sobre el mismo grafo.

Los grafos que se han probado en los experimentos se han generado aleatoriamente. Cada arista recibe un peso entero aleatorio entre 1 y 10. Si el grafo está implementado mediante una lista de adyacencia, su número de nodos es $n \cdot 2^{20}$, donde n es un entero que varía desde 1 hasta 11. El grado de cada nodo es de 7, donde su sucesor se elige aleatoriamente de entre el resto de nodos. Si el grafo está implementado como una matriz de adyacencia, no es posible guardar en memoria la misma cantidad de nodos usada en la versión de listas de adyacencia debido a que su consumo de memoria es alto. Así, el número de nodos es $n \cdot 2^{10}$ nodos, donde n es un entero que varía desde 1 hasta 15. En este caso, el grado de cada nodo es de $n/5$. En ambas implementaciones, los resultados se han obtenido midiendo los algoritmos sobre 25 grafos aleatorios y mediando sus resultados para cada valor de n.

En la figura 2.21 se muestran los resultados para los grafos implementados con listas de adyacencia, y en la figura 2.22 los resultados para los grafos implementados con matrices de adyacencia. En ambas gráficas se aprecia que todas las versiones paralelas en GPU son más rápidas que sus versiones en CPU, incluida la que usa Fibonacci Heaps.

El rendimiento del algoritmo F^{GPU} es mejor que U^{GPU} cuando el grafo está implementado con una lista de adyacencia. Sin embargo, ocurre lo contrario para matrices de adyacencia. Al ser el grado de cada nodo mayor en matrices que en listas, aumenta la probabilidad de que dos nodos frontera requieran relajar la estimación de un mismo nodo. Como consecuencia, F^{GPU}

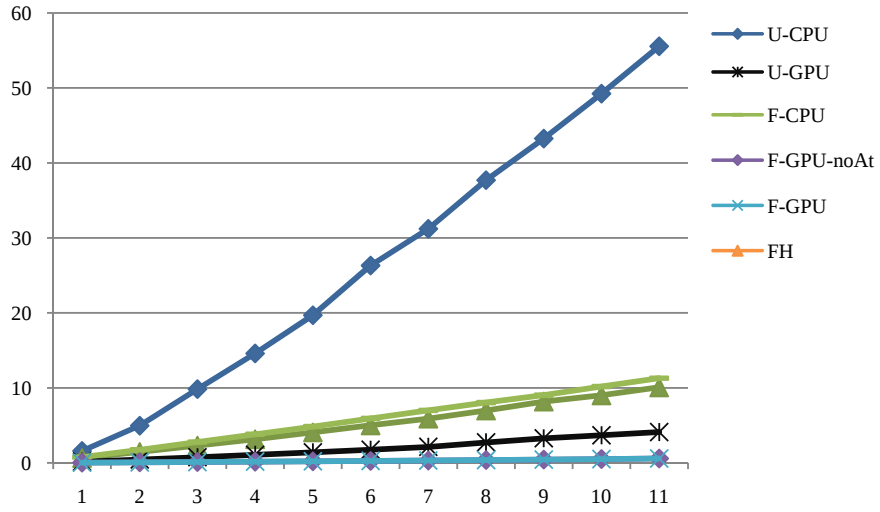


Figura 2.21: Resultados para la lista de adyacencia. En el eje x se encuentra el número de nodos usados en unidades de 2^{20} nodos. En el eje y se encuentra el tiempo en segundos.

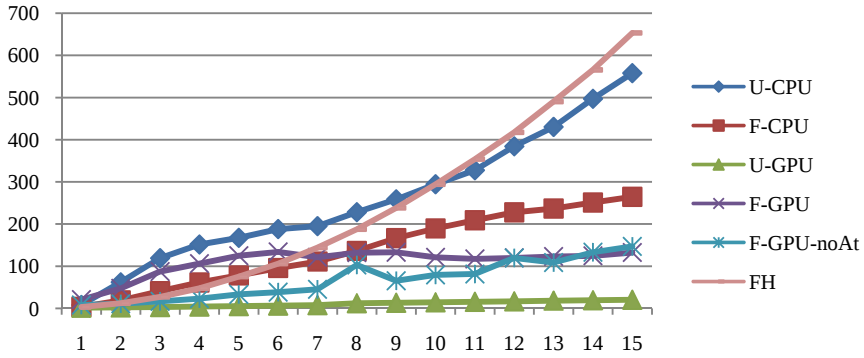


Figura 2.22: Resultados para la matrices de adyacencia. En el eje x se encuentra el número de nodos usados en unidades de 2^{10} nodos. En el eje y se encuentra el tiempo en milisegundos.

aumenta el número de operaciones atómicas y se degrada el rendimiento del kernel. Este hecho se comprueba también observando el comportamiento de $F^{GPU\_no\_Atomic}$. Así, en la versión con listas de adyacencia, el rendimiento de F^{GPU} y $F^{GPU\_no\_Atomic}$ es aproximadamente igual, mientras que en el caso de matrices de adyacencia, $F^{GPU\_no\_Atomic}$ tarda menos tiempo que F^{GPU}, en general.

Capítulo 3

Estructuras de Aceleración

3.1. Introducción

El algoritmo que encuentra el punto de intersección más cercano de cada rayo con la escena es común a todos los algoritmos de ray tracing (capítulo 1). Sea r un rayo definido por su origen o y su dirección d. Decimos que ese rayo r *interseca* al objeto de la escena *obj* si el punto

$$o + t_{hit} \cdot d$$

pertenece al objeto *obj*, para un cierto valor positivo $t_{hit} > 0$. El valor t_{hit} recibe el nombre de *tiempo de intersección* del rayo r con el objeto *obj*. Definimos el conjunto de todas las intersecciones I_r del rayo r con todos los objetos de la escena, como

$$I_r = \Big\{ t_{hit} \in \mathbb{R} \mid (t_{hit} > 0) \wedge (obj \in Escena) \wedge (o + t_{hit} \cdot d \in obj) \Big\}$$

El *tiempo de intersección más cercano* para el rayo con la escena t_{min} se define como

$$t_{min} = min\big(I_r\big)$$

Es posible que un rayo no interseque ningún objeto de la escena, dejando al conjunto I_r vacío. En ese caso, decimos que el tiempo de intersección más cercano del rayo es ∞, indicando que el rayo se ha salido de la escena.

La obtención del punto de intersección más cercano es una de las partes más lentas de los algoritmos de ray tracing. Por ello, cualquier intento de acelerar esta etapa supone un aumento en el rendimiento global. El método directo de obtención del punto de intersección más cercano, llamado *método de fuerza bruta*, consiste en probar secuencialmente la intersección del rayo con todos los objetos de la escena. Desafortunadamente, este método no es realizable en la práctica debido a la gran cantidad de rayos y de objetos involucrados en el proceso de renderizado[1].

Para acelerar la obtención del punto de intersección más cercano, se han desarrollado estructuras de datos llamadas *estructuras de índices* o *de aceleración* (EA, en lo sucesivo). Estas estructuras determinan que ciertos conjuntos de objetos no van a intersecar con un rayo. Por tanto, no será necesario que se realicen los tests de intersección con esos objetos, pudiendo descartarse

[1]Supongamos, por ejemplo, una escena relativamente sencilla formada por $2^{17} = 128$K triángulos. Supongamos también que la imagen final tiene una resolución de 1024×1024 píxeles y que, de media, las rutas tienen una longitud de tres (dos rebotes). Aunque solo se trazase una ruta por píxel, el número de intersecciones rayo-triángulo que se podría llegar a comprobar con el método de fuerza bruta es $3 \cdot 2^{20} \cdot 2^{17} = 3 \cdot 10^{37}$, una cantidad realmente enorme que hace irrealizable este método en la práctica.

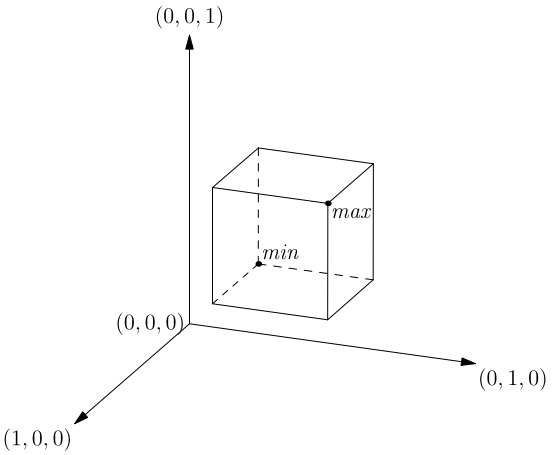


Figura 3.1: Una caja alineada con los ejes o AABB.

con seguridad. Con la integración de estas estructuras, la etapa que encuentra el punto de intersección más cercano se transforma en la etapa en la que cada rayo *recorre* la EA. Por ello, esta etapa recibe el nombre de *etapa de recorrido* o *de traversal*. El recorrido de un rayo por la EA está formado por una serie de pasos en los que cada rayo decide qué partes de la escena *visitar* y cuáles *descartar*, evitando la realización de una gran cantidad de tests de intersección. De ahí el nombre de estas estructuras.

3.2. Bounding Volume Hierarchy

Un *volumen recubridor* (o *bounding volume*) es un objeto de volumen finito que puede ser intersecado por un rayo. Esferas y cilindros son ejemplos de estos volúmenes aunque lo más habitual es el uso de *cajas alineadas con los ejes* (figura 3.1). Una caja alineada con los ejes, también llamada *AABB* (de *Axis-Aligned Bounding Box*) o simplemente *caja*, está formada por seis planos, paralelos dos a dos, y que son también paralelos a los ejes, es decir, son de la forma $x = cte$, $y = cte$ o $z = cte$. Cada par de planos paralelos define el volumen correspondiente al espacio que queda entre ellos. Una caja, por tanto, es el volumen convexo correspondiente a la intersección del espacio que delimita cada pareja de planos paralelos. Una caja está perfectamente especificada por dos puntos *min* y *max*. Así, un punto $p \in \mathbb{R}^3$ pertenece a la caja C si

$$p_x \in [min_x, max_x] \ \wedge \ p_y \in [min_y, max_y] \ \wedge \ p_z \in [min_z, max_z]$$

Nótese que pueden existir infinitos puntos de intersección entre un rayo y una caja. Llamaremos punto de entrada, o t_{entry}, al tiempo de intersección menor (o más cercano) del rayo con el volumen y, análogamente, tiempo de salida, o t_{exit}, al punto de intersección mayor (o más lejano).

Una *Jerarquía de Volúmenes Recubridores* o *BVH* (de *Bounding Volume Hierarchy*) es un árbol, típicamente binario, donde cada nodo tiene asociado un volumen recubridor (figura 3.2). Los nodos hoja, además del volumen de la caja, también mantienen una lista de referencias a triángulos. La característica fundamental de una BVH es que el volumen asociado a un nodo interno contiene a los volúmenes de todos sus hijos. Si el nodo es hoja, entonces su volumen contiene a todos los triángulos referenciados por esa hoja. Esta propiedad es fundamental, ya que en caso de que no existiese intersección entre un rayo y la caja de un nodo, se tiene la garantía de que tampoco existe la intersección con ninguno de sus hijos. Por tanto, todo ese subárbol puede ser descartado completamente durante el recorrido.

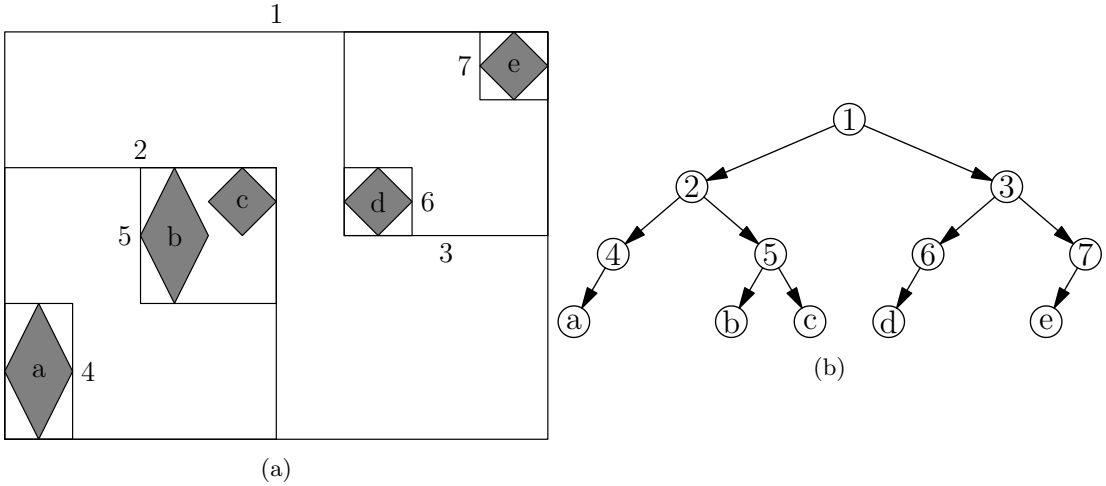


Figura 3.2: Fig. (a). Ejemplo de una escena sobre la que se ha construido una BVH. Los objetos están enumerados con letras mientras que las cajas recubridoras con números. Fig (b). Estructura en árbol de la BVH de la escena de la figura (a).

El algoritmo de recorrido de un rayo por una BVH que se describe a continuación se debe a Aila y Laine [AL09], aunque existen variantes dependiendo de cómo se construya la BVH. Dado un rayo, el algoritmo comienza insertando el nodo raíz en la pila vacía y estableciendo la variable $t_{min} = \infty$. En cada vuelta del bucle, se saca un nodo de la pila. Si el nodo es interno, se prueba la intersección de ese rayo con las cajas de todos sus hijos. Si no existe ninguna intersección, se saca otro nodo de la pila. En caso de que existan múltiples intersecciones, se ordenan los nodos hijo en función de su valor t_{entry} y se apilan comenzando por los nodos más lejanos, quedando el más cercano en la cima. Si el nodo que se saca de la pila es hoja, entonces se prueba la intersección del rayo con todos los triángulos referenciados por la hoja y, en caso de que exista intersección rayo-triángulo, t_{min} se actualiza al valor más pequeño entre t_{min} y t_{hit}. La variable t_{min}, por tanto, mantiene el tiempo de intersección con el triángulo más cercano hasta ese momento.

El algoritmo acaba cuando la pila de recorrido queda vacía. Se podría pensar que, ya que los nodos se ordenan en función de la profundidad del rayo, la primera intersección que se obtiene será ya la más cercana. Eso no es cierto en general porque pueden existir solapamientos entre los nodos (figura 3.3). Es también habitual usar el valor de t_{min} para descartar más subárboles durante el recorrido. En el momento en que se ordenan los hijos de un nodo, se pueden descartar con seguridad aquellos hijos cuyos $t_{entry} > t_{min}$ ya que ningún triángulo del subárbol va a estar más cerca que los ya encontrados. Este descarte recibe el nombre de *early culling*. La ordenación por distancia de los nodos durante el apilado sirve, por tanto, para que esta técnica descarte más nodos durante el recorrido.

Kay y Kajiya [KK86] proponen usar conjuntos de parejas de planos paralelos, llamados *stabs*, como volúmenes recubridores. De forma análoga a las cajas, el espacio interior de un volumen recubridor es la intersección de los espacios que se encuentra entre las parejas de planos paralelos. Además, usan una cola de prioridad, implementada como un montículo, para guardar los nodos pendientes de ser procesados y recuperar el siguiente nodo de recorrido con menor t_{entry}.

Rubin y Whitted [RW80] también usan cajas como volúmenes recubridores, con la diferencia de que estas cajas no siempre están alineadas con los ejes ya que pueden haber sido previamente rotadas. Esto les permite representar cada triángulo de la escena mediante tres cajas. De esta forma, el algoritmo de intersección rayo-caja es el único que necesitan para el renderizado completo,

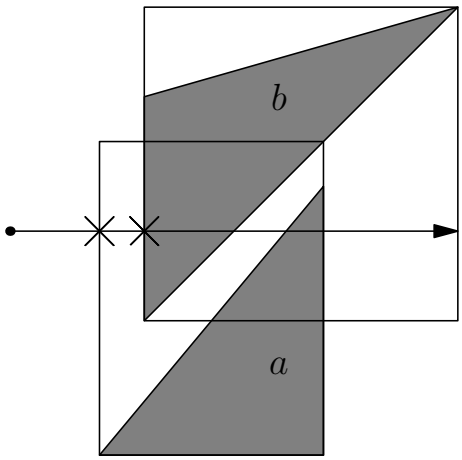


Figura 3.3: Durante el recorrido del rayo, este encuentra primero una intersección con el triángulo a porque su caja se encuentra antes de la caja del triángulo b. Sin embargo, posteriormente el rayo encontrará una intersección más cercana en el triángulo b. Esto es consecuencia del solapamiento de las cajas de los dos nodos en una BVH.

prescindiendo del test rayo-triángulo.

3.3. KD-Tree

Un KD-Tree es un árbol binario. Cada nodo hoja contiene una lista de referencias a triángulos, al igual que una BVH. Sin embargo, cada nodo interno de un KD-Tree contiene un plano alineado con los ejes, es decir, su normal solo puede ser $(1,0,0)$, $(0,1,0)$ o $(0,0,1)$ (figura 3.4). Un plano de estas características está especificado por su dimensión, $k_{pl} \in \{x, y, z\}$, y su posición, p_{pl}. Estos planos dividen el espacio asociado con el nodo en dos partes. Por convenio, el hijo izquierdo es el nodo que se encuentra a la "izquierda" del plano, es decir, aquel nodo cuyos puntos de su volumen son menores que p_{pl} en la dimensión k_{pl} del plano. Simétricamente, el hijo derecho es el que se encuentra a la "derecha" del plano.

En un KD-Tree se cumple que si las coordenadas de todos los vértices de un triángulo en la dimensión k_{pl} son menores que p_{pl} entonces ese triángulo va a estar en alguna hoja de su hijo izquierdo. Análogamente, si los vértices son mayores, el triángulo se encontrará en el subárbol del hijo derecho. Si el triángulo interseca el plano, entonces el triángulo estará referenciado en sus dos hijos.

Sea $caja_{escena}$ la caja más pequeña que encierra toda la escena. Cada plano del KD-Tree divide el volumen de $caja_{escena}$ en dos partes, siendo cada una de ellas nuevamente otra caja. De esa forma, se puede definir implícitamente un volumen para cada nodo del KD-Tree, el que corresponde con la $caja_{escena}$, pero limitada por todos los planos de sus nodos ancestros. De la misma manera, los triángulos referenciados por una hoja corresponden a triángulos que intersecan el volumen implícito de la hoja.

En un KD-Tree, el recorrido termina cuando se encuentra el primer punto de intersección con un triángulo ya que, a diferencia de las BVHs, no existe intersección entre los volúmenes de nodos hermanos. Por lo tanto, un recorrido en profundidad garantiza que la primera intersección corresponde con la intersección más cercana.

Un primer algoritmo de recorrido de un KD-Tree se puede encontrar en la tesis de V. Havran

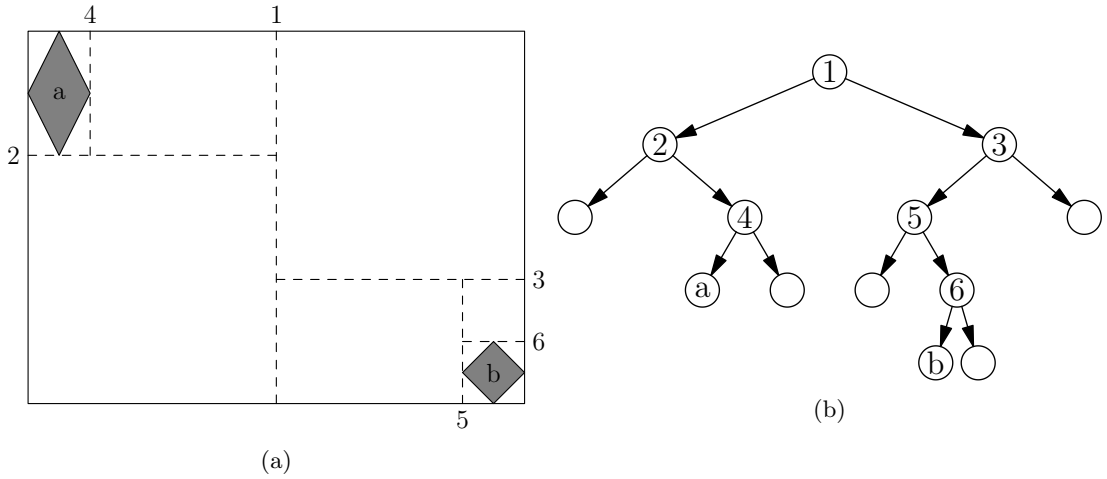


Figura 3.4: Fig. (a). Ejemplo de una escena sobre la que se ha construido un KD-Tree. Los objetos están enumerados con letras mientras que los planos divisores con números. Fig (b). Estructura en árbol del KD-Tree de la escena de la figura (a). Los nodos hoja pueden contener referencias a triángulos o estar vacías.

([Hav00], pág. 94) con el nombre de *recorrido secuencial* (*sequential ray traversal algorithm*, en inglés), basado en el algoritmo que, dado un punto, encuentra la hoja del KD-Tree que lo contiene. Para ello, primero se desciende por la EA hasta encontrar la hoja a la que pertenece el origen del rayo[2] decidiendo en cada nodo interno por qué hijo continuar en función de la posición del propio origen con respecto al plano del nodo. Una vez se ha encontrado la hoja, se realiza la intersección del rayo con todos los triángulos allí referenciados. Si no se encuentra intersección, entonces se calcula el punto de la caja de la hoja por el que sale el rayo. Dicho punto se mueve ligeramente a lo largo de la dirección del rayo para obtener un nuevo origen, y el recorrido vuelve a comenzar desde la raíz.

En el recorrido secuencial de un KD-Tree, es posible que un rayo recorra varias hojas antes de encontrar su punto de intersección más cercano. Ya que, cada vez que no encuentra intersección en una hoja, el recorrido comienza de nuevo desde la raíz, algunos nodos internos serán consultados múltiples veces. Por ello, J. Arvo [Arv88] propone un algoritmo de recorrido recursivo. Nosotros vamos a describir el algoritmo de Havran [Hav00] (págs. 96 y 155–156), que elimina la recursión explícita del algoritmo de Arvo introduciendo una pila.

Para cada rayo se mantiene el segmento de intersección con el volumen del nodo actual de recorrido. Dicho segmento viene especificado por dos valores positivos, t_{ini} y t_{end}. El algoritmo comienza insertando el nodo raíz en la pila vacía e iniciando $t_{ini} = 0$ y $t_{end} = \infty$, es decir, abarcando la totalidad del rayo. En cada paso del recorrido, se saca un nodo de la pila. Si es interno, sus dos hijos se clasifican en *near* y *far* en función de dónde se encuentra el origen del rayo. Si el origen se encuentra a la izquierda del plano, entonces el hijo izquierdo es el nodo *near* y el derecho el *far*. Si el origen se encuentra a la derecha, entonces los papeles se invierten.

Posteriormente, se realiza la intersección del rayo con el plano, obteniendo el tiempo de intersección, t_{pl}. La obtención de este valor se consigue con solo dos operaciones, debido a que el

[2] Si el origen está fuera de la escena, entonces el rayo se corta con la caja de la escena antes de comenzar el recorrido.

plano está alineado con los ejes

$$t_{pl} = \frac{p_{pl} - o[k_{pl}]}{d[k_{pl}]}$$

donde $o[k_{pl}]$ y $d[k_{pl}]$ son las coordenadas k_{pl}-ésima del origen y de la dirección del rayo, respectivamente. La posición relativa de t_{pl} con respecto a t_{ini} y t_{end} indica el orden de recorrido de los hijos del nodo. Si $t_{ini} \leq t_{pl} \leq t_{end}$, entonces se apila el nodo *far* y se prosigue con *near*. Si $t_{pl} < 0$ o $t_{end} < t_{pl}$, solo se tiene que recorrer el nodo *near*. En el último caso, $0 < t_{pl} < t_{ini}$, solo se tiene que recorrer el nodo *far*. En cualquier caso, hay que actualizar el intervalo $[t_{ini}, t_{end}]$, usando t_{pl}, para que este corresponda con el intervalo $[t_{entry}, t_{exit}]$ de la caja implícita del nodo. El intervalo $[t_{entry}, t_{exit}]$ se añade a la pila de recorrido junto con su nodo. El recorrido termina cuando se encuentra el primer punto de intersección o cuando la pila está vacía, lo que significa que el rayo no ha intersecado con ningún objeto de la escena.

Havran et al. [HBv98] presentan un KD-Tree cuyas hojas incorporan seis punteros. Cada puntero, uno por cada cara de su caja asociada, apunta al nodo adyacente a esa hoja que contiene completamente la cara. De esa forma, si un rayo no encuentra intersección en una hoja, solo tiene que calcular la cara por la que sale y seguir el puntero para continuar el recorrido. Este recorrido prescinde del uso de pila aunque el árbol consume más memoria como consecuencia de añadir seis punteros por hoja.

Havran et al. [HKBv97] desarrollan un recorrido con pila en el que los casos más frecuentes se evalúan con menos operaciones, acelerando, por tanto, el proceso completo. Además, soluciona el problema que ocurre cuando el origen del rayo está muy cerca del plano de división debido a los errores de precisión en las operaciones de coma flotante. La solución que proponen consiste en guardar en la pila las coordenadas de los puntos de intersección, en vez de los tiempos del rayo, t_{ini} y t_{end}. Eso permite manejar directamente la posición p_{pl} del plano, en vez del tiempo t_{pl}, que se ha obtenido mediante operaciones con números reales.

Havran y Bittner [HB07] presentan un nuevo recorrido para KD-Trees. A los nodos del KD-Tree situados a ciertas profundidades se les añade explícitamente la caja que tienen asociada. Además, cada nodo aumentado contiene un puntero a su nodo ancestro aumentado. Esto permite que cualquier nodo aumentado pueda servir como nodo de comienzo del recorrido. Los nodos aumentados se aprovechan de dos formas durante el recorrido. Por una parte, para realizar un recorrido sin pila del KD-Tree. Cuando un rayo llega a una hoja y no encuentra intersección, el rayo se acorta y asciende siguiendo los punteros hasta encontrar el primer nodo aumentado cuya caja interseca con el rayo, pudiéndose reanudar el recorrido descendente de nuevo. En segundo lugar, los rayos secundarios pueden comenzar directamente desde el nodo aumentado donde se encuentra su origen, en vez de comenzar desde la raíz.

3.4. Construcción de Estructuras Jerárquicas

Construir una EA para una escena consiste en construir una BVH o un KD-Tree que cumpla las propiedades antes descritas y que contenga al menos una referencia a cada triángulo. Obviamente, existen muchas maneras de construir estructuras jerárquicas para una misma escena, aunque no todas las jerarquías exhiben el mismo rendimiento durante el recorrido. Actualmente, los mejores resultados experimentales se han conseguido siguiendo la construcción *top-down* de MacDonald y Booth [MB90] y usando la *heurística del área de superficie* o *SAH* (*Surface Area Heuristics* en inglés) de Goldsmith y Salmon [GS87].

El algoritmo de construcción top-down es un algoritmo voraz que sigue un esquema divide-y-vencerás. Primeramente, se construye el nodo raíz del árbol teniendo en cuenta todos los triángulos de la escena. Luego, los triángulos se reparten entre sus hijos y se procede recursivamente hasta que

no se realiza ninguna división más, es decir, hasta que el nodo queda como una hoja. El algoritmo de construcción concreto varía ligeramente si consideramos un KD-Tree o una BVH.

En un KD-Tree, un nodo interno queda perfectamente especificado por un plano alineado con los ejes. Primeramente, se construye el nodo raíz eligiendo un plano de división para todos los objetos de la escena. El plano separa los triángulos entre sus dos nodos hijos. Si corta algún triángulo, entonces este será referenciado por sus dos hijos. Las dos listas de referencias a triángulos son tratadas recursivamente igual que se hizo para el nodo raíz.

En una BVH el procedimiento es parecido. En este caso, los objetos son repartidos en dos grupos y se construye la caja más ajustada a los triángulos de cada grupo. Ahora, cada triángulo solo se incluye en uno de los hijos ya que, a diferencia de un KD-Tree, los solapamientos están permitidos. Aunque en una BVH no hay ninguna restricción sobre el número de hijos que puede tener cada nodo interno, la BVH generada por el algoritmo top-down es un árbol binario (*full-binary tree*).

En este procedimiento top-down existen dos puntos que no han sido especificados todavía. Primero, cómo se elige el plano de división, en caso de un KD-Tree, o los dos hijos, en caso de una BVH. Segundo, el *criterio de terminación*, es decir, decidir cuándo mantener una lista de triángulos como hoja o continuar dividiéndola. Para tomar estas dos decisiones es necesario tener una estimación del coste de un árbol, que analizamos a continuación.

3.5. Estimación del Coste de un KD-Tree y una BVH

Para construir *buenas* estructuras de aceleración es necesario definir una función de *coste* sobre dichas estructuras. De esa manera, un buen algoritmo de construcción va a ser aquel que reciba la lista de triángulos de la escena como entrada y devuelva la estructura con menor coste.

Definimos el coste de una estructura jerárquica como el valor $|ray_{root}| \cdot coste(root)$, donde $root$ es la raíz del árbol, ray_i es el conjunto de rayos que intersecan al nodo i durante el renderizado de la escena, es decir, durante el recorrido por la EA, y la función *coste* es el coste medio por rayo de la estructura. La función *coste* está definida recursivamente como

$$coste(n) = \begin{cases} C_t \cdot Tri(n) & \text{si } n \text{ es un nodo hoja} \\ C_i + \frac{|ray_l|}{|ray_n|} coste(l) + \frac{|ray_r|}{|ray_n|} coste(r) & \text{si } n \text{ es un nodo interno} \end{cases} \tag{3.1}$$

Los nodos l y r son los hijos izquierdo y derecho, respectivamente, del nodo n, y $Tri(i)$ es el número de triángulos del nodo hoja i. Obsérvese que $|ray_l|+|ray_r| \geq |ray_n|$, ya que es posible que un mismo rayo tenga que recorrer ambos nodos. C_t es el coste del test de intersección rayo-triángulo y C_i es el coste del test de intersección de un rayo con un nodo interno.

Los parámetros C_t y C_i aportan las unidades de medición a la función de coste. Excepto que se diga otra cosa, consideraremos como coste el número total de tests de intersección que se evalúan cuando se renderiza una escena. Entre las intersecciones contamos las intersecciones rayo-plano (solo en KD-Trees), rayo-caja (solo en BVHs) y rayo-triángulo (en ambas). Por tanto, $C_i = 1$ en un KD-Tree porque representa la intersección rayo-plano que se realiza en cada nodo interno. Si es una BVH, entonces C_i es el número de hijos del nodo. Ya que, en la construcción top-down, los nodos internos tienen siempre dos hijos, entonces $C_i = 2$. El valor C_t es 1 en ambos casos. Una medida alternativa sería considerar C_t y C_i como el tiempo que requiere comprobar cada intersección, en cuyo caso el coste estaría medido en ciclos de reloj o en segundos.

El valor $|ray_{root}|$ se puede omitir porque es el mismo para todas las EA que se llegaran a construir, por tanto, sería suficiente implementar un algoritmo que devolviera el árbol cuya función *coste* fuera mínima. Sin embargo, dicho algoritmo no es factible en la práctica. En primer lugar, para dividir un nodo el algoritmo top-down necesitaría construir todos los posibles subárboles para sus dos hijos, y evaluar los costes de todos ellos. Obsérvese que el número de subárboles que se

pueden formar sobre una lista de triángulos resultaría enorme. En segundo lugar, el número de rayos usados solo se conoce durante el renderizado, lo que implica que el coste de cada estructura se tendría que evaluar tras el proceso completo de renderizado. En definitiva, usar la función de coste según se ha presentado no es factible en la práctica, por lo que se han desarrollado heurísticas que la aproximan.

3.5.1. Heurística del Área de la Superficie

La cantidad de rayos que se usa para renderizar una escena es muy grande. Por tanto, podemos aproximar la relación $\frac{|ray_l|}{|ray_n|}$ mediante la probabilidad condicionada $P(l|n)$ de que un rayo que ha intersecado el nodo n también interseque el nodo l, esto es $\frac{|ray_l|}{|ray_n|} \approx P(l|n)$, y análogamente para el hijo derecho, $\frac{|ray_r|}{|ray_n|} \approx P(r|n)$.

Para derivar la probabilidad $P(l|n)$ sin conocer ni $|ray_l|$ ni $|ray_n|$ usaremos *probabilidad geométrica*, que está determinada como una relación entre funciones de medida geométrica. Para obtener fácilmente una función de medida sobre los rayos, admitimos tres supuestos:

1. Los rayos comienzan fuera de la escena.
2. Los rayos no llegan a bloquearse por objetos de la escena, es decir, no encuentran ninguna intersección durante su recorrido. Por tanto, terminan fuera de la escena.
3. La distribución de los rayos es uniforme.

Aunque estos supuestos no son reales para los rayos que se usan durante el renderizado, sirven para poder derivar con sencillez la probabilidad de intersección de cada nodo, como ahora veremos.

Sea $\mathcal{R}$ el conjunto de todos los rayos que se pueden usar durante el renderizado. De manera similar al capítulo 1, definimos el vector de dirección de un rayo ω como el vector que comienza en el origen de coordenadas y termina en un punto de la superficie de una esfera unidad $\mathcal{S}^2$. Para una dirección ω dada, asumimos que el origen o del rayo se encuentra sobre un plano perpendicular a dicha dirección Π_ω (figura 3.5). La posición del plano Π_ω no es importante siempre y cuando se encuentre fuera de la escena. Además, el plano Π_ω tiene que ser único para cada dirección ω, lo que evita ambigüedades en la especificación de los rayos. Así, el conjunto de rayos $\mathcal{R}$ queda definido como

$$\mathcal{R} = \left\{(o, \omega) \mid \omega \in \mathcal{S}^2 \ \wedge \ o \in \Pi_\omega\right\}$$

Obsérvese que todo rayo de $\mathcal{R}$ cumple trivialmente el supuesto (1).

Definimos el espacio de probabilidad como el conjunto de todos los rayos de $\mathcal{R}$ que intersecan la caja de la escena

$$\mathcal{R}_{root} = \left\{r \in \mathcal{R} \mid r \text{ interseca con } caja_{root}\right\}$$

Los eventos de nuestro espacio de probabilidad son los conjuntos $\mathcal{R}_n$, donde n es un nodo de la estructura de aceleración. También definimos la probabilidad del nodo n, $P(n)$, como la probabilidad de que un rayo aleatorio interseque con su caja asociada $caja_n$, o, dicho de otro modo, la probabilidad de que un rayo pertenezca al evento $\mathcal{R}_n$

$$P(n) = P(\mathcal{R}_n)$$

El supuesto (2) permite derivar una probabilidad para $P(n)$ en la que no se tiene en cuenta otra caja o triángulo de la escena excepto la propia caja de n y la caja de la escena. Esto es consecuencia de que hemos supuesto que ningún rayo puede quedarse bloqueado durante su recorrido, por lo que la probabilidad de intersecar una caja es independiente de los objetos que se encuentren a su alrededor.

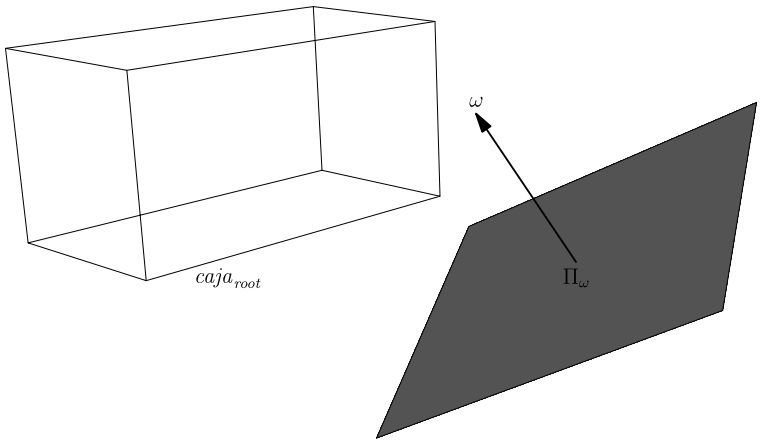


Figura 3.5: Especificación de los rayos bajo los tres supuestos. Todos los rayos con la misma dirección ω tienen sus orígenes en el plano Π_ω, que es perpendicular a la dirección y se encuentra fuera de la caja de la escena.

Sin embargo, el supuesto (2) también implica cierta *simetría* sobre los rayos de un evento $\mathcal{R}_n$. Así, si un rayo con dirección ω interseca la $caja_n$, entonces el rayo con dirección $-\omega$ también lo hará[3]. Para evitar considerar múltiples veces los mismos rayos en el cálculo de la probabilidad de una caja, restringimos el espacio de los rayos a solo aquellos cuyas direcciones pertenezcan al hemisferio canónico $\mathcal{H}$

$$\mathcal{R} = \Big\{(o,\omega) \mid \omega \in \mathcal{H} \ \wedge \ o \in \Pi_\omega\Big\}$$

Recordemos que $\mathcal{H}$ es la semiesfera de radio unidad y centrada en el origen de coordenadas cuyo polo se encuentra en el punto $(0,0,1)$. Obsérvese que ahora $\mathcal{R}$ equivale al conjunto de las rectas en $\mathbb{R}^3$, por lo que hablaremos indistintamente de rectas o de rayos durante la derivación de la probabilidad de intersección de una caja.

El supuesto (3) indica que la densidad de probabilidad de los rayos es constante. Para derivar una pdf sobre los rayos, definimos una medida μ para los eventos. Dicha medida se define a partir del área A y del ángulo sólido σ como

$$\mu(\mathcal{R}_n) = \int_{r\in\mathcal{R}_n} 1\, d\mu(r) = \int_{r\in\mathcal{R}_{root}} \chi_{\mathcal{R}_n}(r)\, d\mu(r) = \int_{\omega\in\mathcal{H}}\int_{o\in\Pi_\omega} \chi_{\mathcal{R}_n}(r)\, dA(o)\, d\sigma(\omega) \tag{3.2}$$

donde $\chi_{\mathcal{R}_n}$ es la función característica del conjunto $\mathcal{R}_n$, es decir, la función es 1 si un rayo pertenece a $\mathcal{R}_n$ y 0 en otro caso. Con esta medida, la pdf sobre los rayos queda como sigue

$$p(r) = \frac{1}{\mu(\mathcal{R}_{root})}$$

para todo rayo $r \in \mathcal{R}_{root}$. La probabilidad de un evento es, por definición, la integral de la pdf sobre todos los rayos del evento

$$P(n) = P(\mathcal{R}_n) = \int_{r\in\mathcal{R}_n} p(r)\, d\mu(r) = \frac{\mu(\mathcal{R}_n)}{\mu(\mathcal{R}_{root})}$$

[3]Sea A un conjunto cualquiera de rayos y $-A$ el conjunto de los mismos rayos de A pero con direcciones de signo opuesto. Si se calcula la probabilidad —se verá más adelante— de ambos conjuntos, se obtienen los mismos resultados. Esta simetría es consecuencia de la simetría de la proyección ortogonal y del supuesto (2).

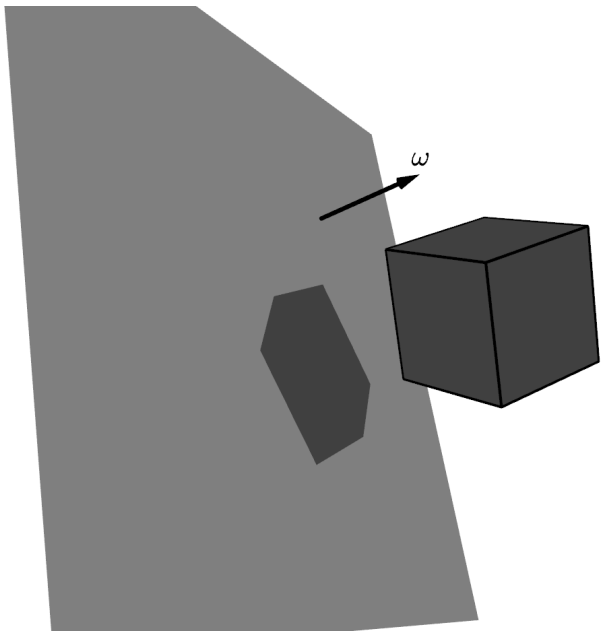


Figura 3.6: Proyección ortogonal de una caja sobre un plano que tiene a ω por normal.

Ahora bien, fijada una dirección ω, el área que abarca la función característica de $\mathcal{R}_n$ es el área de la proyección ortogonal de la $caja_n$ en esa dirección ω (figura 3.6), por ello la ecuación 3.2 se convierte en

$$\mu(\mathcal{R}_n) = \int_{\omega \in \mathcal{H}} A\big(proy\_orth(caja_n, \omega)\big)\, d\sigma(\omega) \tag{3.3}$$

Ya que la $caja_n$ es una AABB, solo una cara de cada par de caras paralelas, como máximo, es visible para los rayos con dirección ω. El área de la proyección ortogonal de una de estas caras se obtiene multiplicando su área por el coseno del ángulo que se forma entre ω y su normal. Si llamamos Δ_x, Δ_y y Δ_z a las áreas de cada cara de cada pareja de caras paralelas, y θ_x, θ_y y θ_z a los ángulos entre ω y la normal de cada cara, entonces el área de la proyección es

$$A\big(proy\_orth(caja_n, \omega)\big) = \Delta_x |\cos\theta_x| + \Delta_y |\cos\theta_y| + \Delta_z |\cos\theta_z|$$

Las normales de las caras son $N_x = (1,0,0)$, $N_y = (0,1,0)$ y $N_z = (0,0,1)$, junto con sus respectivos vectores opuestos, por tanto, la proyección ortogonal se puede expresar, usando el producto escalar de vectores, como

$$A\big(proy\_orth(caja_n, \omega)\big) = |N_x \cdot \omega|\Delta_x + |N_y \cdot \omega|\Delta_y + |N_z \cdot \omega|\Delta_z = |\omega_x|\Delta_x + |\omega_y|\Delta_y + |\omega_z|\Delta_z$$

Desarrollando la ecuación 3.3 tenemos

$$\begin{aligned}
\mu(\mathcal{R}_n) &= \int_{\omega \in \mathcal{H}} \Big(|\omega_x|\Delta_x + |\omega_y|\Delta_y + |\omega_z|\Delta_z \Big)\, d\sigma(\omega) \\
&= 4 \int_0^{\pi/2} \int_0^{\pi/2} \Big(\Delta_x \sin\theta\cos\phi + \Delta_y \sin\theta \sin\phi + \Delta_z \cos\theta \Big) \sin\theta \, d\theta \, d\phi \\
&= \pi(\Delta_x + \Delta_y + \Delta_z) \\
&= \frac{\pi}{2} SA(caja_n)
\end{aligned}$$

donde $SA(caja_n)$ es el área de la superficie de $caja_n$. Así, la probabilidad de intersección del nodo n por un rayo bajo los tres supuestos anteriores es

$$P(n) = P(\mathcal{R}_n) = \frac{\mu(\mathcal{R}_n)}{\mu(\mathcal{R}_{root})} = \frac{SA(caja_n)}{SA(caja_{root})}$$

Esta heurística, que aproxima la probabilidad de intersección de un nodo como una relación entre áreas de superficies, recibe el nombre de *heurística del área de la superficie* o *SAH* (de *Surface Area Heuristics*, en inglés).

La probabilidad de que un rayo interseque el nodo n, sabiendo que ya ha intersecado a su nodo padre m, es la probabilidad condicionada del evento $\mathcal{R}_n$ dado el evento $\mathcal{R}_m$

$$P(n|m) = P(\mathcal{R}_n|\mathcal{R}_m)$$

Con las suposiciones anteriores, si $caja_n \subseteq caja_m$, entonces también se cumple que $\mathcal{R}_n \subseteq \mathcal{R}_m$. Asimismo, también se cumplen las siguientes propiedades

$$\begin{array}{lrcll}
(1) & P(n|m) & = & \frac{P(\mathcal{R}_n \cap \mathcal{R}_m)}{P(\mathcal{R}_m)} = \frac{P(n)}{P(m)} & \text{si } caja_n \subseteq caja_m \\
(2) & P(n|q) & = & \frac{P(n)}{P(q)} = \frac{P(n)}{P(m)}\frac{P(m)}{P(q)} = P(n|m)P(m|q) & \text{si } caja_n \subseteq caja_m \subseteq caja_q \\
(3) & P(n|root) & = & \frac{P(n)}{P(root)} = P(n) &
\end{array}$$

3.5.2. Criterio Automático de Terminación

El caso base en la construcción de un KD-Tree o una BVH se produce cuando una lista de triángulos deja de dividirse y se convierte en un nodo hoja. Existen casos triviales en los que no se puede continuar dividiendo un conjunto de triángulos. En un KD-Tree, no tiene sentido dividir más si la lista de triángulos se ha quedado vacía. En una BVH, no se puede seguir dividiendo si solo existe un triángulo en la lista.

Aparte de estos casos, también se puede establecer un criterio *ad-hoc* para no seguir dividiendo. Por ejemplo, si el número de triángulos de la lista está por debajo de un cierto umbral o si el nodo se encuentra a una cierta profundidad en el árbol. Es posible justificar la introducción de criterios ad-hoc para, por ejemplo, limitar el uso de memoria. Sin embargo, estos umbrales son introducidos por el usuario y no tienen en cuenta la estimación del coste de la estructura.

Otra manera de detener la división de un nodo y, consecuentemente, convertirlo en una hoja, consiste en usar la propia ecuación de coste (ecuación 3.1). Aparte de considerar todas las maneras de dividir un conjunto de triángulos para partir un nodo interno, también se calcula el coste de dejarlo como hoja. Si este coste es menor que el coste de cualquier división, entonces la construcción termina y la lista de triángulos se convierte en un nodo hoja. Ya que de esta manera no es necesario que se establezcan criterios *ad-hoc*, este criterio recibe el nombre de *criterio automático de terminación*.

3.5.3. Consideración del Coste de los Hijos

En la ecuación de coste (ecuación 3.1), los valores $coste(l)$ y $coste(r)$ no son conocidos. Como ya se ha comentado, construir todos los posibles árboles y devolver el de coste mínimo no es factible en la práctica, por lo que tienen que ser nuevamente aproximados. La aproximación que se usa consiste en suponer que ambos hijos son hojas, y que, por tanto, su coste es el producto de C_t por el número de triángulos. Esto, en general, es una cota superior del coste de la división del nodo, ya que lo más probable es que cada hijo no se quede como una hoja sino que sea posteriormente dividido (porque el coste de la división sea menor que dejar al nodo como hoja), sobre todo para los nodos cercanos a la raíz.

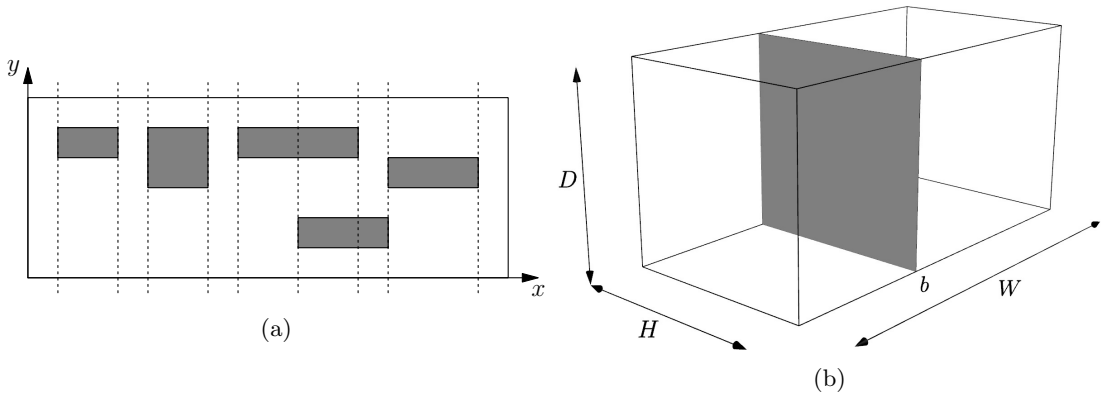


Figura 3.7: Fig. (a). Planos candidatos perfectos sobre el eje x. Fig. (b). División de una caja por un plano candidato. Los valores W, H y D indican las dimensiones de la caja. El valor de b indica el porcentaje con respecto a W de la posición del plano.

3.5.4. Estimación del Coste de una Estructura

Dada una división de la lista de triángulos de un nodo, podemos calcular una estimación del coste que tendría ese nodo en el árbol final mediante las suposiciones anteriores. Así, la función $\hat{coste}$, que aproxima la función *coste* (ecuación 3.1) queda como

$$\hat{coste}(n) = C_i + C_t \frac{SA(caja_l)}{SA(caja_n)} Tri(l) + C_t \frac{SA(caja_r)}{SA(caja_n)} Tri(r) \tag{3.4}$$

donde C_t es siempre igual a 1, y C_i es 1 o 2, dependiendo de si se está construyendo un KD-Tree o una BVH, respectivamente. A esta función $\hat{coste}$ se le llama *estimación de coste* o, simplemente, *coste SAH*.

El algoritmo top-down de construcción de una EA para una escena consiste, por tanto, en probar todas las divisiones posibles de un conjunto de triángulos. Para cada posible división se evalúa su coste SAH así como el coste SAH de dejar todo como una hoja. Si el mínimo de todos los costes es el coste de dejarlo como una hoja, entonces la lista de triángulos se convierte en un nodo hoja. Si el mínimo está en una división, entonces se elegirá esa división y se procederá recursivamente con los dos hijos. Los detalles concretos de qué divisiones se consideran son diferentes para KD-Trees (sección 3.6) y BVHs (sección 3.7).

3.6. Algoritmo Top-down con Coste SAH para KD-Trees

3.6.1. Planos Candidatos Perfectos

En un KD-Tree, un *plano candidato* es cualquier plano que pueda servir como plano divisor de un nodo. Como cualquier plano alineado con los ejes es un plano candidato en un KD-Tree, podría pensarse que el número de ellos es infinito. Sin embargo, MacDonald y Booth [MB90] demostraron que solo es necesario considerar aquellos planos para los que cambia el número de triángulos repartidos a cada lado. En otras palabras, el coste SAH mínimo se encuentra en alguno de los planos que contienen los límites de las cajas de los triángulos (figura 3.7a). Estos planos reciben el nombre de *planos candidatos perfectos*. MacDonald y Booth demostraron los dos siguientes hechos.

1) La función de coste SAH es monótona creciente, decreciente o constante en los lugares en los que el número de triángulos no cambia. Supongamos, sin pérdida de generalidad, que solo vamos a dividir un nodo por su eje x (figura 3.7b). Siguiendo la notación del artículo original [MB90], consideremos la siguiente función

$$f(b) = SA_l(b)N_l(b) + SA_r(b)N_r(b)$$

donde f es el coste SAH (ecuación 3.4), pero eliminando los valores que no dependen de b. El parámetro $b \in [0,1]$ indica el porcentaje del ancho de la caja donde se coloca el plano candidato. El resto de valores son entonces: $SA_l(b) = 2HD + 2b\,WD + 2b\,WH$ y $SA_r(b) = 2HD + 2(1-b)\,WD + 2(1-b)\,WH$ son el área de la superficie de la parte izquierda y derecha de la caja; $N_l(b)$ y $N_r(b)$ son el número de triángulos a la izquierda y a la derecha del plano candidato. Trabajar con f es equivalente a hacerlo con $\hat{coste}$ ya que sus mínimos se encuentran en el mismo plano candidato.

La función f es continua en aquellos intervalos en los que el número de triángulos no cambia. Por tanto, su comportamiento se puede obtener a partir de la primera derivada respecto de b

$$f'(b) = SA'_l(b)\,\big(N_l(b) - N_r(b)\big)$$

ya que

$$SA'_l(b) = -SA'_r(b) = 2\,WD + 2\,WH$$

y $N'_l(b) = N'_r(b) = 0$ cuando el número de triángulos no cambia. Como se puede observar, la derivada de f es constante (positiva, negativa o cero) cuando ni N_l ni N_r cambian. Por tanto, la función es monótona creciente, decreciente o constante a trozos, y es en los lugares en los que cambia el número de triángulos donde se producen las discontinuidades.

2) En una discontinuidad, el coste SAH es el mínimo de los dos valores posibles. Supongamos, sin pérdida de generalidad, el caso de la figura 3.8. En este ejemplo, el objeto obj_2 se encuentra a la izquierda del plano candidato, mientras que a su derecha no se encuentra ningún objeto hasta el objeto obj_3. El valor b_0 representa la proporción (sobre el total) de la posición del plano candidato. Consideremos ahora solo el intervalo que existe desde el límite derecho de obj_1 hasta el límite izquierdo de obj_3. Para valores de b mayores que b_0, la función f tiene que considerar un elemento menos en N_r que para valores de b menores. El número de objetos a la izquierda no cambia. Sean f_1 y f_2 dos funciones que representan a f antes y después de b_0. Los límites de f por la derecha y por la izquierda en b_0 corresponden a los valores de f_1 y f_2 en b_0

$$\begin{array}{rcccl} \lim\limits_{b\to b_0^-} f(b) & = & f_1(b_0) & = & SA_l(b_0)\,N_l(b_0) + SA_r(b_0)\,(N_r(b_0)+1) \\ \lim\limits_{b\to b_0^+} f(b) & = & f_2(b_0) & = & SA_l(b_0)\,N_l(b_0) + SA_r(b_0)\,N_r(b_0) \end{array}$$

deduciéndose que $f_2(b_0) < f_1(b_0)$. El coste SAH —y, por tanto, f— considera que el objeto no pertenece a la parte derecha, por lo que $f(b_0)$ es el mínimo de los dos valores, $f(b_0) = f_2(b_0) = min\big(f_1(b_0), f_2(b_0)\big)$. El caso en el que b_0 se encuentra en el límite izquierdo del objeto es simétrico.

Conclusión. Por 1) y 2) se concluye que el plano candidato que minimiza la función de coste SAH se encuentra en los planos candidatos que recogen los limites de las cajas de los triángulos.

3.6.2. Algoritmo de Construcción $O(N \log N)$

Para la construcción de KD-Trees, hemos seguido el algoritmo de Wald y Havran [WH06], que implementa la construcción top-down de KD-Trees en $O(N \log N)$, donde N es el número de triángulos de la escena. Además, este algoritmo maneja como caso especial los triángulos coplanarios a un plano candidato perfecto, es decir, aquellos triángulos cuyos vértices se encuentran

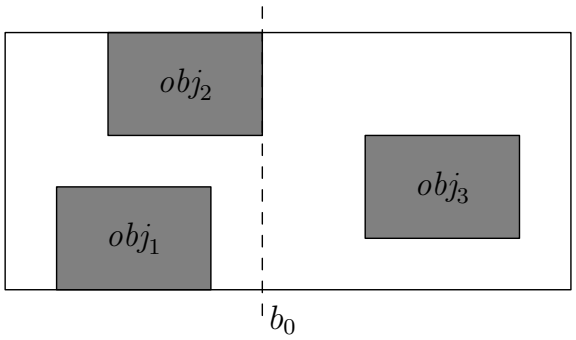


Figura 3.8: El plano candidato se sitúa sobre el límite derecho de la caja del objeto obj_2.

sobre un plano candidato. Estos triángulos reciben el nombre de *triángulos planos*. Esta clase de triángulos no son una excepción en las escenas tridimensionales porque aparecen siempre que se consideran objetos alineados con los ejes, tales como paredes o suelos.

Definimos un *evento*[4] como una tupla de la forma (ε, k, id, t), donde ε es la posición del plano candidato perfecto en la dimensión k, id es el identificador del triángulo que generó el evento, y t es el tipo del evento. Los eventos pueden ser de tres tipos, *start*, *end* y *plane*, identificados con los símbolos $+$, $-$ y $|$, respectivamente. Cada triángulo puede generar uno o dos eventos en cada dimensión. Si el triángulo con identificador id es plano en la dimensión k, entonces genera el evento *plane* $(caja.min[k], k, id, |)$. Si no es plano en la dimensión k, entonces genera dos eventos, el evento *start* $(caja.min[k], k, id, +)$ y el evento *end* $(caja.max[k], k, id, -)$.

El algoritmo top-down sigue un esquema divide-y-vencerás sobre la lista de eventos. Para que el coste del algoritmo sea $O(N \log N)$, se realiza una ordenación de los eventos al principio del algoritmo que se mantienen ordenados en sucesivas llamadas recursivas. El orden entre los eventos, similar al orden lexicográfico, es el siguiente

$$\begin{aligned}(\varepsilon_1, k_1, id_1, t_1) < (\varepsilon_2, k_2, id_2, t_2) \quad \equiv_{def} \quad & (k_1 < k_2) \ \vee \ (k_1 = k_2 \ \wedge \ \varepsilon_1 < \varepsilon_2) \ \vee \\ & (k_1 = k_2 \ \wedge \ \varepsilon_1 = \varepsilon_2 \ \wedge \ t_1 < t_2)\end{aligned}$$

donde el orden de los tipos se ha establecido como primero los end, luego los plane y finalmente los start $(- < | < +)$. Lo más importante de este orden es que los eventos que caen sobre un mismo plano candidato (eventos con k y ε iguales) aparecerán contiguos en la lista y ordenados por su tipo. Como veremos ahora, este orden sirve para obtener el coste SAH de todos los planos candidatos iterando una sola vez sobre la lista de eventos.

Debido a que los eventos están ordenados por posición, es fácil obtener las variables p^+, p^- y $p^|$ de cada plano candidato, que representan el número de triángulos que comienzan (p^+) y terminan (p^-) en cada plano candidato, o son coplanario $(p^|)$ con él. Para obtener el coste SAH de un plano candidato, calculamos el número de triángulos que están a su izquierda (N_L), a su derecha (N_R), o son coplanarios (N_P) a partir de p^+, p^- y $p^|$, recorriendo la lista de eventos. Estos valores se pueden obtener de manera incremental de la siguiente manera

$$\begin{aligned} N_L^{(0,k)} &= 0 & \qquad N_L^{(i,k)} &= N_L^{(i-1,k)} + p_{i-1}^| + p_{i-1}^+ \\ N_P^{(0,k)} &= p_0^| & \qquad N_P^{(i,k)} &= p_i^| \\ N_R^{(0,k)} &= Tri - p_0^| & \qquad N_R^{(i,k)} &= N_R^{(i-1,k)} - p_i^| - p_i^- \end{aligned}$$

[4]Nótese que este *evento* no tiene relación con un suceso posible en probabilidad, donde también se llama evento. Se ha mantenido el nombre de "evento" porque es el que usan los autores en su artículo.

donde i es el i-ésimo plano candidato en la dimensión k, y *Tri* es el número total de triángulos.

Los tres valores $N_L^{(i,k)}$, $N_P^{(i,k)}$ y $N_R^{(i,k)}$ se usan para obtener el número de triángulos a cada lado del plano candidato i-ésimo en la dimensión k y poder así evaluar su coste SAH. Los triángulos planos pueden incluirse en el hijo izquierdo o en el derecho y, por tanto, deben considerarse las dos posibilidades.

Cuando se han terminado de procesar los eventos, se ha evaluado el coste SAH de todos los planos candidatos perfectos. Al plano con coste SAH mínimo lo denotamos como $(\hat{\varepsilon}, \hat{k}, \hat{side})$, donde $\hat{\varepsilon}$ es la posición del plano en la dimensión $\hat{k}$, y $\hat{side}$ indica a qué lado se tienen que añadir los N_P triángulos planos (si los hubiera).

Una vez obtenido el plano con menor coste SAH, hay que repartir los triángulos y los eventos entre sus dos hijos. Para ello recorremos nuevamente los eventos y clasificamos cada triángulo como *LeftOnly*, *RightOnly* y *Both*, en función de si se encuentra, respectivamente, completamente a la izquierda, completamente a la derecha o se extiende entre los dos hijos (corta el plano). Un triángulo se encuentra completamente a la izquierda de un plano si se cumple alguna de las siguientes condiciones:

1. No es plano y termina antes o sobre el plano. Entonces, existe un evento (ε, k, id, t) tal que

$$(t = -) \wedge (\varepsilon \leq \hat{\varepsilon}) \wedge (k = \hat{k})$$

2. Es plano y está antes que el plano. Entonces, existe un evento (ε, k, id, t) tal que

$$(t = |) \wedge (\varepsilon < \hat{\varepsilon}) \wedge (k = \hat{k})$$

3. Es plano y está sobre el propio plano. En este caso, irá en el hijo izquierdo si así lo indica la variable $\hat{side}$. Por tanto, existe un evento tal que

$$(t = |) \wedge (\varepsilon = \hat{\varepsilon}) \wedge (\hat{side} = \mathit{left}) \wedge (k = \hat{k})$$

Los triángulos completamente a la derecha tienen propiedades simétricas. Si un triángulo no está clasificado de alguna de las maneras anteriores entonces se extiende por ambos hijos y se clasifica como *Both*. De esta manera, los triángulos ya se pueden repartir entre sus dos hijos.

Una vez repartidos los triángulos, hay que obtener las nuevas listas de eventos que generarían los triángulos de cada hijo. La manera directa sería generar los eventos de nuevo a partir de los triángulos de cada hijo, pero esto implicaría necesariamente ordenarlos de nuevo, lo que aumentaría el coste asintótico del algoritmo. Otra forma más eficiente consiste en usar la lista de eventos previa. Por un lado, los eventos que son generados por los triángulos etiquetados como *LeftOnly* o *RightOnly* no cambian. Por tanto, se recorre nuevamente la lista de eventos, se comprueba la etiqueta del triángulo que generó cada evento y se añade a una de las listas de eventos de cada hijo, llamadas E_{LO} y E_{RO}, según corresponda. Los eventos de estas dos listas ya están ordenados, por lo que no es necesario volverlos a ordenar.

Por otro lado, los triángulos etiquetados como *Both* cortan el plano de división, por lo que generan nuevos eventos que se guardan en las listas E_{BL} (para el hijo izquierdo) y E_{BR} (para el hijo derecho). Esos eventos son generados en cualquier orden, por lo que tienen que ordenarse posteriormente. Sin embargo, el coste de su ordenación es menor que $O(N \log N)$ ya que se estima que el número de triángulos cortados por un plano divisor es del orden de $O(\sqrt{N})$, resultando en un coste de $O(\sqrt{N} \log \sqrt{N}) \subset O(\sqrt{N} \times \sqrt{N}) = O(N)$. Por último, se mezclan las listas ya ordenadas E_{LO} con E_{BL}, y E_{RO} con E_{BR}, obteniéndose los eventos de cada hijo. El algoritmo continúa recursivamente en cada hijo hasta que se cumple alguno de los criterios de terminación antes explicados.

3.7. Algoritmo Top-down con Coste SAH para BVHs

La construcción de una BVH es más flexible que la de un KD-Tree, ya que existen $O(2^N)$ maneras diferentes de repartir los N objetos de un nodo entre sus dos hijos. Sin embargo, calcular el coste SAH de todas estas formas es, de nuevo, imposible en la práctica por lo que la construcción se ha simplificado para hacerla parecida a la de un KD-Tree. Para la construcción de BVHs que hemos seguido en esta tesis hemos utilizado el algoritmo de construcción ampliamente usado debido a Wald et al. [WBS07].

En la construcción de la BVH, solo los centros de las cajas de los triángulos, llamados centroides, son tenidos en cuenta durante la clasificación. En cada dimensión, se coloca un plano alineado con los ejes sobre cada centroide. Ese plano divide a los objetos en dos grupos, que corresponden con los dos hijos de un nodo. Así, si hay C centroides en la lista, entonces existen $C-1$ divisiones posibles en cada dimensión. Posteriormente, se calcula la caja más ajustada a cada grupo de triángulos y se evalúa el coste SAH de la división. Al igual que con los KD-Trees, se elige la división con menor coste SAH, incluyendo la comparación con el coste SAH de dejar el nodo como hoja.

Para calcular el coste SAH de cada división de manera eficiente, el algoritmo de construcción ordena los centroides en una dimensión. Posteriormente, la lista ordenada se recorre de izquierda a derecha para obtener incrementalmente, en esa dimensión, todas las cajas más ajustadas del hijo izquierdo. Se repite el paso anterior recorriendo la lista de derecha a izquierda y obteniendo la caja más ajustada del hijo derecho. Una vez terminados los dos recorridos se calcula el coste SAH de todas las divisiones. El proceso se repite para cada dimensión y se obtiene la división con coste SAH mínimo. El algoritmo de construcción tiene coste $O(N \log^2 N)$ ya que se realiza una ordenación $O(N \log N)$ en cada llamada recursiva.

I. Wald [Wal07] evita la ordenación en cada paso obteniendo un algoritmo con coste $O(N \log N)$. Para ello, primero divide cada dimensión en espacios equidistantes llamados *bins*. Posteriormente, cada centroide se clasifica en función de su bin, tarea que se puede realizar en tiempo constante debido a que los espacios son equidistantes. Este paso, por tanto, sustituye un paso de coste $O(N \log N)$ por uno de coste $O(N)$. Cada bin guarda el número de triángulos cuyos centroides caen dentro junto con la caja más ajustada de dichos triángulos. Al igual que antes, para calcular el coste SAH se necesita recorrer la lista de bins de izquierda a derecha y viceversa.

Popov et al. [PGDS09] proponen un algoritmo de construcción de BVHs más general que el de Wald et. al [WBS07], pero sin llegar a evaluar todas las maneras de repartir los objetos entre los dos hijos, disminuyendo el coste desde $O(2^N)$ a polinomial. Esto les permite obtener BVHs con costes SAH menores aunque su rendimiento en la práctica es menor. Los autores concluyen que el coste SAH solo es una buena estimación cuando las cajas de los nodos hermanos no solapan. Para solucionar este problema introducen un factor de corrección que tiene en cuenta el volumen de intersección entre cajas de nodos hermanos

$$\hat{coste}(n) = C_i + \left(C_O \frac{V(caja_l \cap caja_r)}{V(caja_n)} + 1 \right) \cdot C_t \, \frac{SA(caja_l)Tri(l) + SA(caja_r)Tri(r)}{SA(caja_n)}$$

donde $V(n)$ es el volumen del nodo n y C_O es un parámetro que indica la importancia de la corrección. Empíricamente, los autores llegan a la conclusión de que la mejor manera de construir BVHs es evitando el solapamiento entre cajas de nodos hermanos.

Ernst y Greiner [EG07] se dieron cuenta de que las cajas de las hojas de una BVH desperdician mucho espacio ya que no se ajustan bien a la geometría cuando los triángulos de la escena son grandes. Esto se debe a que la construcción clásica de BVHs, a diferencia de los KD-Trees, no permite divisiones de triángulos. Por ello, los autores relajaron la propiedad de las BVHs de que cada triángulo solo podía ser referenciado por una hoja. Concretamente, presentaron el método *Early Split Clipping*, que divide por la mitad la caja más ajustada de cada triángulo hasta que su

área esté por debajo de un cierto umbral. De esta forma, la construcción de una BVH no cambia excepto en que se usan las cajas de los triángulos previamente divididas, en vez de directamente las cajas de los triángulos.

En el artículo no se propone ningún método para determinar dicho umbral. En la práctica, se tiene que acudir al método de "prueba y error" para determinarlo: si el umbral es demasiado grande no se obtiene beneficio, mientras que si es demasiado pequeño, la BVH crece demasiado y decae su rendimiento.

Dammertz y Keller [DK08] presentan una nueva heurística para dividir los triángulos antes de la construcción de una BVH. Esta heurística, llamada *Edge Volume Heuristic*, construye una caja en cada una de las aristas de cada triángulo. Para cada una de estas cajas, se calcula su volumen y, si está por encima de un cierto umbral, se genera un nuevo vértice en el medio de la arista, quedando dividido el triángulo en dos. Similar al *Early Split Clipping* de Ernst y Greiner [EG07], la idea detrás de este algoritmo es dividir los triángulos cuyas cajas no ajustan bien, es decir, aquellos triángulos cuyas aristas son muy "diagonales".

Stich et al. [SFD09] proponen considerar tanto la división espacial como la objetual para la construcción de BVHs. En la división objetual los planos candidatos se eligen de la misma manera que lo hacen Wald et al. [WBS07]. En la división espacial, los planos candidatos se reparten equidistantemente en la caja del nodo. Esta última forma permite la división de la caja de los objetos, similar a *Early Split Clipping* o *Edge Volume Heuristic*. El plano de división elegido será aquel plano candidato con menor coste SAH.

3.8. Variaciones de la Función de Coste

Como se vio en la sección 3.5.1, para la derivación del coste SAH (ecuación 3.4) se han hecho tres suposiciones sobre los rayos usados durante el renderizado, que pueden no cumplirse totalmente. Esto ha llevado a que se propongan otras heurísticas que adaptan la SAH a condiciones más realistas. Por otro lado, derivar una estimación del coste considerando un conjunto más restrictivo de rayos permite suponer que la EA estará "más especializada" en esos rayos y, por tanto, será más eficiente para dichos rayos durante su recorrido. Sin embargo, las EAs especializadas deben demostrar su eficacia experimentalmente ya que el algoritmo top-down no garantiza un árbol con un coste menor. En la práctica, estas nuevas heurísticas obtienen mejores EAs comparadas con la SAH clásica.

Fabianowski et al. [FFD09] desarrollan una heurística suponiendo que los orígenes de los rayos se encuentran dentro de la escena. Para los rayos secundarios, esta suposición es más aceptable ya que estos se generan en la superficie de los objetos. Debido a que los orígenes de los rayos pueden encontrarse en cualquier punto de la escena, surge de manera natural el uso del volumen como función de medida para los orígenes. La probabilidad de intersección, queda, por tanto, como

$$P(A|B) = \frac{V(caja_A)}{V(caja_B)} + \frac{1}{4\pi V(caja_B)} \int_{o \in (caja_B \setminus caja_A)} \sigma_o(caja_A)\, dV(o)$$

donde A y B son nodos, V es la medida de volumen y $\sigma_o(caja_A)$ es el ángulo sólido que genera la $caja_A$ vista desde el origen o. Desgraciadamente, esa integral no se puede resolver analíticamente y tiene que ser aproximada. Los autores presentan dos métodos de aproximación, uno menos preciso, pero más rápido, y otro más preciso, pero más lento.

W. Hunt [Hun08] presenta una variación de la SAH en la que se tiene en cuenta la técnica del *mail-boxing*. Durante el recorrido de un rayo por un KD-Tree, es posible que se calcule múltiples veces la intersección de ese rayo con un mismo triángulo. Con esta técnica, los últimos tests se guardan en el *mail-boxing*, que actúa como una caché de intersecciones. Sin embargo, el coste SAH original no refleja esta posible disminución en el número de intersecciones. Por ello, el autor resta al

coste SAH el valor $C_{L\wedge R}P_{L\wedge R}$ para incorporar la corrección debida al uso del mail-boxing, donde $C_{L\wedge R}$ es el número de triángulos que solapan con el plano de división, y $P_{L\wedge R}$ es la probabilidad de que un rayo interseque los dos hijos.

Havran y Bittner [HB99] presentan tres variaciones de la SAH obtenidas restringiendo el conjunto de rayos. La primera heurística `(a)` solo considera rayos que tienen la misma dirección. La probabilidad de intersección de una caja es aproximada como el área de la superficie de la caja tras haber sido proyectada ortogonalmente en la dirección común de los rayos. Las heurísticas `(b)` y `(c)` consideran que todos los rayos tienen el mismo origen. En `(b)`, las direcciones de los rayos forman un frustum, por lo que la probabilidad se aproxima como el área de la proyección perspectiva de la caja. En `(c)`, las direcciones no están restringidas por lo que la probabilidad se aproxima como el ángulo sólido subtendido por la caja.

Ninguna de las heurísticas anteriores puede derivarse directamente a partir de la medida de los rayos de la ecuación 3.2 ya que se puede comprobar que los conjuntos de rayos de `(a)`, `(b)` y `(c)` tienen medida cero. Sin embargo, la heurística `(a)` puede obtenerse si solo se tiene en cuenta el área de los orígenes, y la `(c)`, si solo se tiene en cuenta el ángulo sólido. En `(b)`, los autores decidieron usar el área de la proyección perspectiva de la caja en lugar del ángulo sólido del frustum. Ambas medidas no son equivalentes pero son igualmente válidas en cuanto a que cada una implica una suposición diferente sobre la probabilidad de las direcciones de los rayos[5].

Bittner y Havran [BH09] presentan la heurística *RDH* (de *Ray Distribution Heuristics*). En esta heurística, los valores $\frac{|ray_l|}{|ray_n|}$ y $\frac{|ray_r|}{|ray_n|}$ de la ecuación de coste 3.1 no son aproximados con probabilidad geométrica, sino mediante un subconjunto de todos los rayos usados para renderizar. A dicho conjunto se le llama conjunto de rayos representativo o *RRS* (de *Representative Ray Set*). Utilizar un RRS tiene el problema de que es posible que todos los rayos intersequen ambos hijos, sobre todo en niveles cercanos a la raíz. De esa forma, se obtiene el mismo coste para todas las divisiones y, consecuentemente, la construcción resulta en un árbol degenerado. Para solucionarlo, el coste de la división se obtiene ponderando los costes de SAH junto con esta nueva heurística RDH.

Experimentalmente, esta técnica da buenos resultados cuando se considera el algoritmo ray casting. Sin embargo, para otros algoritmos de ray tracing no se obtienen mejores rendimientos. Los autores argumentan que los rayos que cumplen los tres supuestos de la SAH representan mejor a los rayos usados durante del renderizado que los rayos del conjunto RRS.

V. Havran en su tesis doctoral ([Hav00], pág. 63) argumenta que, para un KD-Tree, es más eficiente que el número de hojas vacías sea el mínimo posible. Así, siempre que un rayo tenga que superar un espacio vacío hasta encontrar su punto de intersección más cercano, el recorrido será más rápido si tiene que recorrer menos hojas vacías.

Una forma de obtener pocas hojas con mucho espacio vacío, en vez de muchas hojas con poco espacio vacío, es favorecer la creación de estas hojas vacías lo antes posible durante la construcción del KD-Tree, lo que el autor llama *Early Cutting Off Empty Space*. Para implementar esta técnica, Wald y Havran [WH06] favorecen los planos candidatos que generan hojas vacías, multiplicando su coste SAH por el factor $\lambda = 0.8$.

Wald et al. [WGS04] proponen una heurística que adapta el coste SAH, originalmente desarrollado para ray tracing, a consultas en el photon mapping. La heurística recibe el nombre de *Voxel Volume Hierarchy* o *VVH*. En photon mapping, la irradiancia en un punto se aproxima calculando la densidad de los k fotones más cercanos. Para acelerar esta búsqueda, se construye un KD-Tree sobre todos los fotones. Este KD-Tree tiene la cualidad de que cada nodo lleva asociado, además

[5] Un ejemplo clásico de esto se encuentra en la *paradoja de Bertrand* [Chu83]. El problema consiste en encontrar la probabilidad de que una cuerda de circunferencia elegida aleatoriamente sea mayor que el lado del triángulo equilátero inscrito en dicha circunferencia. Diferentes suposiciones sobre la forma en que se escogen las cuerdas infieren diferentes probabilidades.

del plano divisor, un fotón coplanario con él. Al igual que ocurre con el ray tracing, la forma en que se construye el KD-Tree influye en el rendimiento de cada consulta. Inspirado en el coste SAH, el coste VVH para un nodo interno n es

$$\hat{coste}(n) = 1 + P(l|n)\, Pho(l) + P(r|n)\, Pho(r)$$

donde $Pho(l)$ y $Pho(r)$ son el número de fotones del hijo izquierdo y derecho, respectivamente.

En este caso, la probabilidad $P(l|n)$ es la probabilidad de que el hijo l sea consultado durante la búsqueda de fotones, sabiendo que su padre n ya ha sido consultado. Suponiendo que el origen de las consultas puede ser cualquier punto y que su radio máximo es un valor fijo r_{max}, entonces la probabilidad de consultar un hijo es equivalente a la probabilidad de que una esfera de radio r_{max} interseque con el volumen del nodo. Por tanto, los autores aproximan esa probabilidad mediante

$$P(l|n) \approx \frac{Vol(caja_l \pm r_{max})}{Vol(caja_n \pm r_{max})}$$

donde Vol es la medida de volumen y $(caja_l \pm r_{max})$ es la extensión de la caja una distancia r_{max} en cada dimensión.

Ize et al. [IWP08] usan la heurística del área de la superficie para construir KD-Trees cuyos planos no tienen la restricción de estar alineados con los ejes. Estos KD-Trees reciben el nombre de árboles *BSP* (de *Binary Space Partitioning Trees*). Surgen tres problemas con el cálculo del coste SAH aplicado a BSPs. El primero es que se puede usar cualquier orientación para los planos candidatos, por lo que el número de candidatos es infinito. La solución que proponen los autores es acortar este número de forma que cada triángulo solo genere 10 planos candidatos: los seis alineados con los ejes de su caja, el plano donde reposa el triángulo, y los tres planos ortogonales que pasan por las aristas del triángulo. El segundo problema consiste en calcular rápidamente la superficie del volumen de un nodo que resulta al ser cortado por un plano no alineado con los ejes. Esa superficie se aproxima por el área de la caja que lo contiene. El último problema se debe a los errores de precisión que aparecen cuando se determina en qué lado de los planos candidatos reside cada triángulo. Los autores proponen manejar esas imprecisiones mediante un umbral de tolerancia.

Los resultados experimentales muestran que los BSPs resultantes requieren un menor número de intersecciones, sobre todo en los contornos de los objetos. Esto se debe a que la mayor libertad a la hora de posicionar planos permite un mayor ajuste de la EA sobre los objetos.

Hunt y Mark [HM08a] derivan la probabilidad de intersección de una caja si los rayos poseen la siguiente forma: los orígenes se sitúan sobre un rectángulo en el plano $z = 0$, cuyos lados tienen longitud $2A_x$ y $2A_y$; las direcciones son vectores que comienzan en el origen de coordenadas y señalan a un punto en el plano $z = 1$. La transformación en perspectiva de los objetos de la escena y de los rayos permite a los autores derivar la siguiente probabilidad para los nodos n y m

$$P(n|m) = \frac{G(n)}{G(m)} \qquad \text{donde} \qquad G(n) = \Delta'_z + \frac{A_y}{2}\Delta'_y + \frac{A_x}{2}\Delta'_x$$

donde Δ'_x, Δ'_y y Δ'_z son las áreas de cada par de caras de la caja recubridora del nodo n en el espacio de la proyección perspectiva. Estas suposiciones sobre los rayos se ajustan mejor a los rayos primarios o a los rayos de sombra sobre un área de luz que las suposiciones de la SAH original.

Popov et al. [PGSS06] proponen situar los planos candidatos de división equidistantes. De esta forma, en una primera pasada, cada triángulo puede calcular rápidamente en qué intervalos comienza y termina su caja recubridora. Al terminar esta pasada, se sabe el número de triángulos que han empezado y terminado en cada intervalo. Esta información se usa en una segunda pasada para obtener el número de triángulos que se encuentran en cada intervalo, pudiéndose evaluar el coste SAH de cada plano candidato.

Este procesamiento de los planos candidatos y de los triángulos, junto con una construcción primero-en-anchura, se usa para obtener un buen patrón de acceso a memoria, disminuyendo consecuentemente el tiempo de construcción en CPU. Además, a partir de un cierto nivel de profundidad, la construcción cambia a primero-en-profundidad para aprovechar la memoria caché.

V. Havran, en su tesis doctoral ([Hav00] págs. 72–75), introduce en el coste SAH para KD-Trees la distinción entre rayos *hit* (rayos que intersecan algún triángulo dentro de un nodo) y rayos *miss* (no intersecan ningún triángulo). Así, aparecen tres nuevos elementos en el coste SAH: la probabilidad de que un rayo termine dentro de un subárbol, el coste de un subárbol si el rayo es hit (*coste hit*) y el coste si es miss (*coste miss*). La probabilidad, llamada *blocking factor*, se aproxima proyectando las cajas de los objetos en cada plano alineado con los ejes y calculando el área de la unión. El autor comenta dos inconvenientes para el uso práctico de esta ecuación de coste. La primera es el tiempo extra requerido para calcular la unión del área proyectada por los objetos. La segunda es la dificultad de aproximar el coste hit y el coste miss de un subárbol.

Havran y Bittner [HB02] proponen un criterio de terminación alternativo para la construcción de KD-Trees. Una vez elegido el plano divisor de un nodo, se comprueba si la división de ese nodo hace disminuir el coste del KD-Tree construido hasta ese momento. En caso de que el coste disminuya, la división se acepta y se procede recursivamente. Si el coste aumenta, la división se puede aceptar siempre y cuando las F_{max} divisiones anteriores no hayan aumentado también el coste del KD-Tree. Los autores proponen una ecuación para calcular F_{max} en función de la profundidad máxima que puede tomar el KD-Tree. En caso de que la división no se acepte, el nodo se mantiene como hoja.

Havran et al. [HHS06] presentan una estructura llamada *SKD-Tree*, donde cada nodo interno contiene dos planos paralelos, en vez de solo uno, como en los KD-Trees. Usando una construcción basada en *buckets*, se consiguen tiempos mejores en la construcción de SKD-Trees que en la de KD-Trees, aunque introducen imprecisiones que afectan al rendimiento. Para solucionarlo presentan una estructura híbrida entre un KD-Tree, un SKD-Tree y una BVH, llamada *H-Tree*. Cada nodo del H-Tree puede contener un plano divisor (como en los KD-Trees), dos planos (como en los SKD-Trees), una caja (como en las BVHs) o dos planos acotadores (llamados *slab*, en inglés). La función de coste se modifica para considerar el coste de cualquiera de los nodos anteriores y se elige aquel tipo de nodo que obtenga menor coste SAH.

3.9. Heurísticas Especializadas

La derivación de la heurística del área de la superficie parte de considerar que los rayos usados para el renderizado se pueden aproximar como rectas distribuidas de forma uniforme (sección 3.5). Aceptar otras suposiciones sobre este conjunto de rayos conduce al desarrollo de estimaciones de coste diferentes de la SAH original. En el trabajo de Torres et al. [TMGA12] hemos cambiado dos suposiciones. La primera consiste en no considerar todas las posibles direcciones, sino solo aquellas pertenecientes a un cierto conjunto. La segunda consiste en suponer que no todos los rayos son igualmente probables, sino que algunos se van a dar con más frecuencia que otros.

Restringir el conjunto de rayos permite derivar estimaciones de coste más específicas. Es de esperar que las EAs construidas con estas nuevas estimaciones de coste sean más eficientes para aquellos rayos de su correspondiente conjunto, aunque este hecho se tiene que comprobar experimentalmente (sección 3.9.7).

3.9.1. Heurística del Área de la Superficie para una Sección Esférica

Una forma de restringir el conjunto de rayos es limitar sus direcciones. Si mantenemos el resto de suposiciones, los rayos, al igual que sucedía con la SAH, pueden aproximarse mediante rectas.

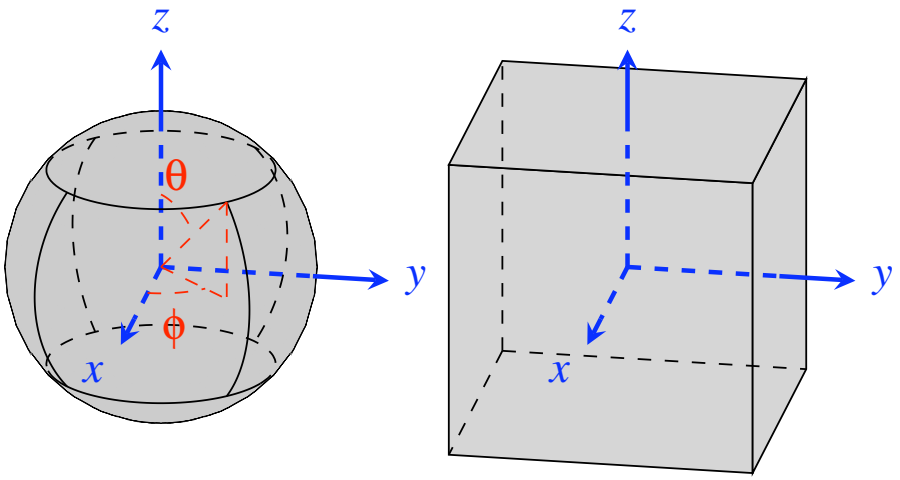


Figura 3.9: Distribución de las secciones esféricas (a la izquierda) y de las secciones cúbicas (a la derecha). Por claridad, se muestran las seis secciones en las dos figuras, aunque solo tres de ellas se consideran para aproximar la probabilidad en la función de coste.

Sin embargo, los vectores directores de estas rectas ya no serán puntos sobre el hemisferio completo, sino solo sobre un subconjunto de él. Obsérvese que esta restricción no afecta a los orígenes de los rayos y que, por tanto, pueden situarse en cualquier lugar, siempre y cuando cumplan las otras suposiciones, es decir, que se encuentren fuera de la escena y en un plano perpendicular a su dirección.

Sea $T : [0, \pi] \times [0, 2\pi] \rightarrow \mathcal{S}^2$ la función que transforma coordenadas esféricas a cartesianas

$$T(\theta, \phi) = (\sin\theta \cos\phi,\ \sin\theta \sin\phi,\ \cos\theta)$$

Sean $\Theta \subset [0, \pi]$ y $\Phi \subset [0, 2\pi]$ dos intervalos en el espacio de las coordenadas esféricas. Si aplicamos T a un rectángulo de la forma $\Theta \times \Phi$, entonces se obtiene una sección de la esfera, a la que llamaremos *sección esférica* o *SP* (de *Spherical Patch*). Ya que los rayos se aproximan como rectas, el rectángulo $\Theta \times \Phi$ tiene que elegirse de tal forma que no se encuentren las direcciones ω y $-\omega$ sobre el mismo *SP*.

En nuestro trabajo de Torres et al. [TMGA12] hemos probado tres diferentes secciones esféricas, denotadas como SP_x, SP_y y SP_z (figura 3.9, a la izquierda). Los intervalos Θ y Φ que han generado estas secciones esféricas se pueden consultar en la tabla 3.1. La constante θ_0 se ha establecido para que la superficie de todos los patches sea la misma, resultando en $\theta_0 = acos(\frac{2}{3})$.

Para definir una medida sobre un conjunto de rayos y, por tanto, derivar la probabilidad de que la caja de un nodo n sea intersecada, es suficiente con restringir el dominio de integración en la ecuación 3.3 a una sección esférica. Cada sección esférica SP_x, SP_y y SP_z da lugar a una nueva medida para un conjunto de rayos, denotadas como $\mu_{orth}^{SP_x}$, $\mu_{orth}^{SP_y}$ y $\mu_{orth}^{SP_z}$, respectivamente

$$\mu_{orth}^{SP_i}(\mathcal{R}_n) = \int_{\omega \in SP_i} A\big(proy\_orth(caja_n, \omega)\big)\, d\sigma(\omega) \tag{3.5}$$

donde $i \in \{x, y, z\}$. La estimación del coste derivada de esta medida recibe el nombre de SPHERE-ORTH. El resultado de la integración de la expresión 3.5 da como resultado tres diferentes pesos, w_x, w_y y w_z, para cada una de las áreas de las caras de la $caja_n$, Δ_x, Δ_y y Δ_z. Por tanto, la probabilidad condicionada de que el nodo n sea intersecado es

$$P(n) = \frac{\mu_{orth}^{SP_i}(\mathcal{R}_n)}{\mu_{orth}^{SP_i}(\mathcal{R}_{root})} = \frac{w_x\Delta_x + w_y\Delta_y + w_z\Delta_z}{w_x\Delta'_x + w_y\Delta'_y + w_z\Delta'_z}$$

Sec. Esférica	**Límites (coord. esféricas)**		**SPHERE-ORTH**			**SPHERE-OBLI**		
	Θ	Φ	w_x	w_y	w_z	w_x	w_y	w_z
SP_x	$[\theta_0, \pi - \theta_0]$	$[\frac{-\pi}{4}, \frac{\pi}{4}]$	55.04	22.80	22.15	53.47	23.59	22.92
SP_y	$[\theta_0, \pi - \theta_0]$	$[\frac{\pi}{4}, \frac{3\pi}{4}]$	22.80	55.04	22.15	23.59	53.47	22.92
SP_z	$[0, \theta_0]$	$[0, 2\pi]$	22.04	22.04	55.90	22.66	22.66	54.67
Sec. Cúbica	**Límites (coord. cartesianas)**			**CUBE-ORTH**			**CUBE-OBLI**	
	x	y	z	w_x	w_y	w_z	w_x	w_y	w_z
CP_x	$\{1\}$	$[-1, 1]$	$[-1, 1]$	51.29	24.35	24.35	50.00	25.00	25.00
CP_y	$[-1, 1]$	$\{1\}$	$[-1, 1]$	24.35	51.29	24.35	25.00	50.00	25.00
CP_z	$[-1, 1]$	$[-1, 1]$	$\{1\}$	24.35	24.35	51.29	25.00	25.00	50.00

Tabla 3.1: Intervalos y pesos normalizados para las heurísticas esféricas y cúbicas. Los valores w_x, w_y and w_z son los pesos normalizados en tanto por ciento para las áreas de las caras Δ_x, Δ_y y Δ_z, respectivamente.

donde Δ'_x, Δ'_y y Δ'_z son las áreas de las caja de la escena. Ya que la probabilidad está definida como una relación entre medidas, esos pesos se pueden normalizar dividiendo numerador y denominador por su suma total. Los valores de los pesos normalizados w_x, w_y y w_z para cada *SP* se pueden consultar en la tabla 3.1. Obsérvese cómo la cara que se encuentra enfrente de cada *SP* (por ejemplo, la cara Δ_x para SP_x) recibe un peso mayor que las demás.

3.9.2. Heurística del Área de la Superficie para una Sección Cúbica

Una manera alternativa de especificar el espacio de direcciones (basada en el trabajo de Hunt y Mark [HM08a]) es usar un cubo centrado en el origen (figura 3.9, a la derecha). Para simplificar los resultados, suponemos que los puntos mínimo y máximo del cubo son los puntos $(-1, -1, -1)$ y $(1, 1, 1)$. De esta forma, las direcciones de los rayos se especifican como los vectores que van desde el origen hasta un punto de la superficie de este cubo. Estos vectores no tienen módulo uno en general, así que tienen que ser normalizados para que puedan usarse como vectores directores de los rayos. Ya que estas direcciones se sitúan en la cara de un cubo, una medida natural para las direcciones es el área en la superficie del cubo.

Usando como referencia este cubo, el espacio de direcciones se puede restringir si consideramos solo las direcciones pertenecientes a una de sus caras. Cada una de esas caras recibe el nombre de *sección cúbica* o *CP* (de *Cubic Patch*). Al igual que sucede con SPHERE-ORTH, cada recta sería considerada dos veces si pudiera representarse con dos direcciones de signo opuesto. Por lo tanto, solo una cara de cada pareja de caras paralelas puede usarse como dominio de integración. Por sencillez, hemos elegido las caras cuya coordenada constante es 1, a las que llamamos CP_x, CP_y y CP_z (tabla 3.1).

Con esta división del espacio de direcciones se pueden derivar tres medidas $\mu_{orth}^{CP_x}$, $\mu_{orth}^{CP_y}$ y $\mu_{orth}^{CP_z}$ que resultan, a su vez, en tres estimaciones del coste. Por ejemplo, para $\mu_{orth}^{CP_z}$ hay que resolver la siguiente integral

$$\mu_{orth}^{CP_z}(\mathcal{R}_n) = \int_{-1}^{1}\int_{-1}^{1} A\left(proy\_orth\left(caja_n, \frac{(x, y, 1)}{\sqrt{x^2 + y^2 + 1}}\right)\right) dx\, dy$$

Este heurística recibe el nombre de CUBE-ORTH y los pesos resultantes de cada cara se pueden consultar en la tabla 3.1.

	COS-ORTH			COS-OBLI		
β	w_x	w_y	w_z	w_x	w_y	w_z
1	43.99	28.00	28.00	33.33	33.33	33.33
2	50.00	25.00	25.00	43.99	28.00	28.00
3	54.08	22.95	22.95	50.00	25.00	25.00
4	57.14	21.42	21.42	54.08	22.95	22.95
5	59.55	20.22	20.22	57.14	21.42	21.42
10	67.01	16.49	16.49	65.90	17.04	17.04

Tabla 3.2: Pesos normalizados en tanto por ciento para la heurística del coseno, tomando algunos valores de β. Sólo presentamos el caso para N_x. Los otros casos pueden obtenerse intercambiado adecuadamente los valores de las columnas.

3.9.3. Probabilidad de los Rayos Ponderada con el Coseno

Otra de nuestras propuestas en cuanto a la relajación de las suposiciones sobre los rayos es considerar que todos los rayos son posibles, aunque no igualmente probables. En concreto, hemos cambiado la densidad de probabilidades de un rayo para que sea proporcional a una potencia del coseno del ángulo que se forma entre la dirección del rayo y un vector D dado

$$p(r) = \frac{|D \cdot \omega|^\beta}{\displaystyle\int_{r \in \mathcal{R}_{root}} |D \cdot \omega|^\beta \, d\mu(r)} \tag{3.6}$$

La constante β es un real positivo que sirve para aumentar la probabilidad de aquellos rayos cuyas direcciones se encuentran más cerca de D. Si $\beta = 0$ entonces la pdf de los rayos es la misma que en la SAH original.

La probabilidad de un conjunto de rayos se puede expresar, al igual que en las heurísticas anteriores, como una relación entre medidas. En este caso, usaremos la siguiente medida μ^D_{orth}, que se expresa en función de μ como

$$\mu^D_{orth}(\mathcal{R}_n) = \int_{r \in \mathcal{R}_n} |D \cdot \omega|^\beta \, d\mu(r)$$

De esta manera, la densidad de probabilidades de la ecuación 3.6 se puede expresar también como

$$p(r) = \frac{|D \cdot \omega|^\beta}{\mu^D_{orth}(\mathcal{R}_{root})}$$

y la probabilidad de que un rayo interseque la caja de un nodo n es

$$P(n) = P(\mathcal{R}_n) = \int_{r \in \mathcal{R}_n} p(r) \, d\mu(r) = \frac{\mu^D_{orth}(\mathcal{R}_n)}{\mu^D_{orth}(\mathcal{R}_{root})}$$

Hemos usado tres vectores para D: $N_x = (1,0,0)$, $N_y = (0,1,0)$ y $N_z = (0,0,1)$. Las tres medidas que se derivan de los vectores anteriores, $\mu^{N_x}_{orth}$, $\mu^{N_y}_{orth}$ y $\mu^{N_z}_{orth}$, respectivamente, definen tres estimaciones de coste, a las que hemos llamado COS-ORTH. Sus pesos normalizados se pueden consultar en la tabla 3.2. Obsérvese que, debido a la elección de los vectores D, solo es necesario obtener los pesos para una de las medidas, los pesos del resto de medidas son permutaciones de los ya obtenidos.

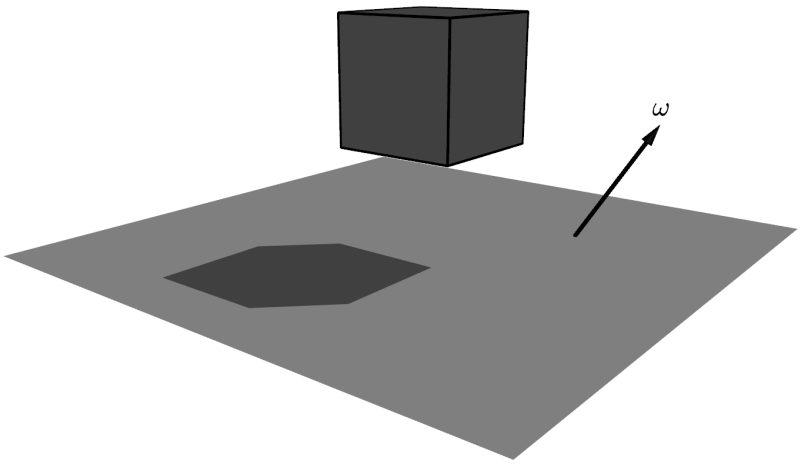


Figura 3.10: Proyección oblicua de una caja sobre un plano en la dirección ω.

3.9.4. Proyección Oblicua

Una recta se puede especificar mediante una dirección y un origen. Según la sección 3.5.1, la dirección se especifica como un punto sobre el hemisferio, mientras que el origen es un punto fuera de la escena que yace sobre un plano ortogonal a la dirección. Una manera alternativa de especificar los rayos consiste en suponer que los orígenes se sitúan todos sobre un mismo plano, independientemente de las direcciones, por lo que se elimina la condición de que este plano sea perpendicular a la dirección elegida. Esta forma de especificar rectas da lugar a que la medida de los orígenes sea el área de la proyección oblicua de la caja sobre el plano (figura 3.10), en vez del área de la proyección ortogonal.

Sin embargo, esta representación tiene dos inconvenientes. Por un lado, no se pueden representar rectas cuyas direcciones sean paralelas al plano elegido. Afortunadamente, este conjunto tiene medida nula y su eliminación no alteraría el valor de la nueva medida. Por otro lado, la superficie de la caja proyectada oblicuamente se hace cada vez mayor a medida que las direcciones de las rectas se alejan de la normal. Como consecuencia, el valor de la medida tiende a infinito y no puede derivarse la probabilidad de que una caja sea intersecada por un rayo.

Lo anterior solo ocurre cuando se considera todo el espacio de direcciones con los supuestos de la SAH original. Sin embargo, sí es posible usar la proyección oblicua cuando también admitimos los supuestos de SPHERE-ORTH, CUBE-ORTH y COS-ORTH. En SPHERE-ORTH y CUBE-ORTH, el conjunto de direcciones está más restringido, por tanto, para derivar una medida, es suficiente con elegir un plano para los orígenes de los rayos de tal manera que no existan direcciones paralelas a dicho plano. En concreto, se ha elegido el plano alineado con los ejes YZ para SP_x y CP_x, el plano XZ para SP_y y CP_y, y el plano XY para SP_z y CP_z.

La proyección oblicua de una caja sobre los planos alineados con los ejes se calcula de forma muy sencilla. Para cada cara, primero se proyecta ortogonalmente la cara y posteriormente se divide por una de las coordenadas del vector de proyección. Por ejemplo, para proyectar la caja del nodo n sobre el plano XY usando la dirección ω obtenemos

$$proy\_obli_{XY}(caja_n, \omega) = \left|\frac{\omega_x}{\omega_z}\right| \Delta_x + \left|\frac{\omega_y}{\omega_z}\right| \Delta_y + \Delta_z$$

En COS-ORTH, el conjunto de direcciones son todas las posibles, sin embargo, no todos los rayos tienen el mismo peso. Esto permite que las medidas derivadas sean finitas. Se han elegido también para esta heurística, los planos alineados con los ejes para realizar la proyección oblicua de las cajas. En concreto, se ha elegido XY para N_z, XZ para N_y, y YZ para N_z.

Las nuevas medidas que se derivan de esta forma de especificar los rayos se obtienen siguiendo el procedimiento de las secciones 3.9.1, 3.9.2 y 3.9.3, pero cambiando la proyección ortogonal por la oblicua. Para estas heurísticas hemos usado la misma notación, sustituyendo el sufijo ORTH por OBLI. Así, se obtienen nueve nuevas medidas, denotadas por $\mu_{obli}^{SP_x}$, $\mu_{obli}^{SP_y}$, $\mu_{obli}^{SP_z}$, para SPHERE-OBLI, $\mu_{obli}^{CP_x}$, $\mu_{obli}^{CP_y}$, $\mu_{obli}^{CP_z}$, para CUBE-OBLI, y $\mu_{obli}^{N_x}$, $\mu_{obli}^{N_y}$, $\mu_{obli}^{N_z}$, para COS-OBLI. En las tablas 3.1 y 3.2 figuran los pesos resultantes.

3.9.5. Uso de Múltiples Estructuras de Aceleración Especializadas

Usar un conjunto de rayos más restrictivo que el dominio completo para estimar la probabilidad de intersección de un nodo (lo que consiguen las secciones esféricas y cúbicas de las heurísticas SPHERE o CUBE) permite derivar la estimación del coste de una EA (ecuación 3.1) solo para un conjunto de rayos. Si esta nueva estimación del coste se usa en el algoritmo top-down de la sección 3.4, es posible construir EAs especializadas para esos conjuntos de rayos. Por tanto, cuando un rayo que pertenece a este conjunto más restrictivo recorre una EA construida de esta manera, el número de nodos recorridos que requiere para encontrar su intersección más cercana será menor en media. Hemos aplicado las heurísticas anteriormente descritas, SPHERE-ORTH, SPHERE-OBLI, CUBE-ORTH y CUBE-OBLI, en la construcción de KD-Trees.

Durante el renderizado de una escena, los rayos involucrados pueden tomar cualquier dirección. Si solo se usase un único KD-Tree especializado, aquellos rayos cuyas direcciones pertenecieran al *SP* o *CP* correspondiente obtendrían un recorrido más rápido, mientras que los rayos cuyas direcciones estuvieran fuera requerirían más tiempo. Experimentalmente, hemos comprobado que el tiempo total requerido es mayor si se usa un único KD-Tree especializado en vez de uno construido con SAH. Por tanto, resulta imprescindible disponer de suficientes KD-Trees especializados durante el renderizado para cubrir el espacio completo de direcciones. Por la forma en que se ha dividido el espacio de direcciones en SPHERE y CUBE, solo es necesario construir tres KD-Trees. Al conjunto de estos tres KD-Trees lo hemos llamado *Multi-KD-Tree*.

El recorrido de un Multi-KD-Tree por un rayo consta de dos fases. En la primera, la dirección del rayo se usa para seleccionar uno de los tres KD-Trees. Si el Multi-KD-Tree se ha construido con CUBE, entonces la selección se parece a la que se usa para seleccionar una cara en *cube mapping*. En concreto, se selecciona el KD-Tree asociado con CP_i si la coordenada i-ésima es la que un tiene mayor valor absoluto de entre las tres coordenadas de la dirección del rayo. Si ha sido construido con SPHERE, entonces primero se comprueba si el valor absoluto de la componente z de la dirección es mayor que $\cos(\theta_0)$. En caso afirmativo, se selecciona el KD-Tree asociado con SP_z. En caso contrario, se comparan los valores absolutos de las coordenadas x e y, y se elige el KD-Tree asociado con la que tiene mayor valor absoluto. En la segunda fase, el KD-Tree seleccionado se recorre usando cualquier algoritmo de recorrido de KD-Trees.

Las heurísticas COS son diferentes ya que consideran todas las direcciones posibles. Experimentalmente, hemos comprobado que, de manera similar a lo que ocurre con SPHERE y CUBE, usar un único KD-Tree construido con COS supone un empeoramiento en el rendimiento en comparación con un KD-Tree construido con la SAH original. Vamos pues a usar la misma solución que con las heurísticas SPHERE y CUBE, es decir, un Multi-KD-Tree de tres KD-Trees. Como ya se ha comentado, cada uno de los tres KD-Trees está construido estableciendo el vector D como una de las direcciones de los tres ejes: $D = (1, 0, 0)$, $D = (0, 1, 0)$ y $D = (0, 0, 1)$. La manera de seleccionar cada KD-Tree del Multi-KD-Tree es la misma que la usada en SPHERE, es decir,

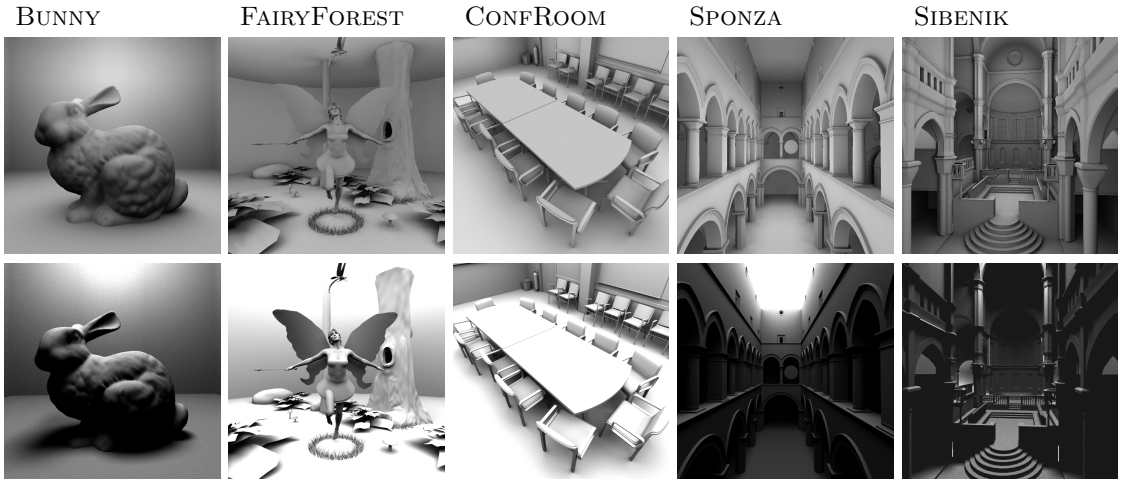


Figura 3.11: Capturas de las escenas usadas en los experimentos. La primera fila muestra las escenas renderizadas con Ambient Occlusion y la segunda con Path Tracing.

mediante la selección del *SP* en función de la dirección del rayo.

3.9.6. Detalles de Implementación en GPU de un Multi-KD-Tree

Hemos implementado en CUDA un Path Tracing (PT) y un Ambient Occlusion (AO) como algoritmos de ray tracing para probar el rendimiento de un Multi-KD-Tree construido según las heurísticas anteriores. Las escenas usadas en nuestros tests son Bunny, FairyForest, ConfRoom, Sponza and Sibenik (figura 3.11). A la escena FairyForest le hemos añadido un techo y a la escena Bunny una caja para evitar que los rayos se salgan de la escena. De lo contrario, muchos rayos primarios se saldrían de la escena, disminuyendo el número de rayos secundarios. Las imágenes generadas tienen una resolución de 1024×1024 y todas las superficies tienen un material diffuse.

Antes del comienzo del renderizado, todos los KD-Trees se construyen en CPU y se cargan en la memoria global de la GPU. El tiempo invertido en la construcción de cada KD-Tree con nuestras heurísticas es aproximadamente el mismo que requiere la SAH original. Los nodos de cada KD-Tree de un Multi-KD-Tree se guardan en el mismo array, llamado *nodes*, de manera que los nodos pertenecientes al mismo KD-Tree se guardan en secciones consecutivas. En el array *references* se guardan las referencias a los triángulos de todos los nodos hoja, y los índices de los nodos raíz se guardan en el array llamado *header*.

La tabla 3.3 muestra el número de nodos y la memoria consumida por un KD-Tree basado en la SAH y un Multi-KD-Tree construido con SPHERE-ORTH. Como se puede observar, la memoria consumida por el Multi-KD-Tree es aproximadamente el triple del espacio requerido por el KD-Tree basado en la SAH. Los requisitos del resto de las heurísticas son similares y no se han incluido.

Detalles de Implementación del Path Tracing

Para este algoritmo vamos a considerar solo dos niveles de recursión: rayos *primarios* y *secundarios*. El algoritmo está compuesto por tres kernels: generador de rayos (o *RG*, de *Ray Generator*), recorrido (o *TI*, de *Traversal-Intersection*), y shading (o *SH*). El diagrama de flujo de los kernels CUDA se puede consultar en la figura 3.12 a la izquierda. Debemos precisar que este es un PT implícito, es decir, que no se lanzan rayos de sombra desde los puntos de intersección hacia

		KD-Tree con SAH			SPHERE-ORTH		
Escena	Triángulos	Num.Nodos	Num.Ref.	Memoria	Num.Nodos	Num.Ref.	Memoria
BUNNY	69,475	536,639	343,082	9.49 MB	1,738,331	1,092,768	30.69 MB
FAIRYFOREST	174,119	1,257,457	922,883	22.70 MB	3,983,961	2,901,640	71.85 MB
CONFROOM	282,761	1,570,225	1,433,336	29.42 MB	5,253,325	4,723,711	98.17 MB
SPONZA	67,464	436,899	367,534	8.06 MB	1,339,641	1,141,669	24.79 MB
SIBENIK	80,143	358,779	311,503	6.66 MB	1,100,537	965,394	20.47 MB

Tabla 3.3: Datos de un KD-Tree construido con SAH y un Multi-KD-Tree construido con SPHERE-ORTH. *Triángulos* es el número de triángulos de cada escena. *Num.Nodos* es el número de nodos (tanto internos como hojas) de los KD-Trees. *Num.Ref.* es el número total de referencias a triángulos. *Memoria* es el consumo de memoria (cada nodo consume 16 bytes y cada referencia 4 bytes).

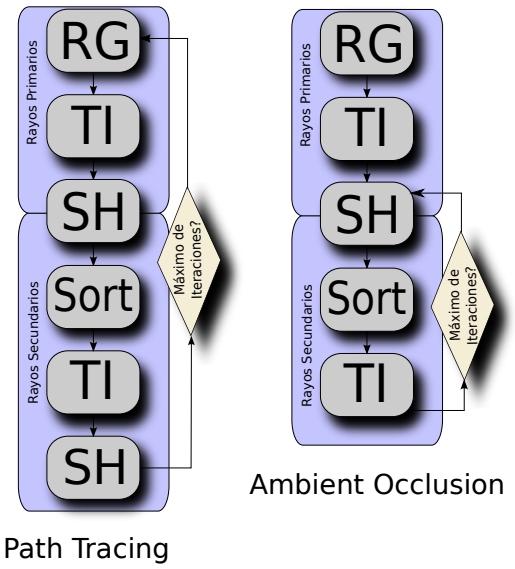


Figura 3.12: Diagrama de flujo de los kernels de Path Tracing (a la izquierda) y Ambient Occlusion (a la derecha).

las luces. Para completar la imagen final son necesarias varias iteraciones de los kernels.

El kernel *RG* genera los rayos primarios desde la cámara (una cámara *pinhole*). En cada iteración, se generan cuatro muestras aleatorias por píxel, lanzando al recorrido un lote de 4MRays ($= 4 \cdot 2^{20}$ rayos). En este kernel, cada rayo elige el KD-Tree que debe recorrer, tal y como se describió en la sección 3.9.5. El kernel *TI* se encarga de encontrar el punto de intersección más cercano para cada rayo. Este kernel es realmente el algoritmo *persistent while-while* de Aila y Laine [AL09], pero adaptado a KD-Trees (este algoritmo de recorrido se describirá en la sección 4.3). Antes de comenzar el algoritmo, cada rayo consulta el array header para conocer la posición del nodo raíz del KD-Tree seleccionado en la etapa anterior. El kernel *SH* genera los rayos secundarios a partir de los rayos primarios y acumula el color de la ruta en el buffer de la imagen final. De manera similar a lo que sucede en *RG*, es en este kernel donde los nuevos rayos secundarios eligen el KD-Tree que debe recorrer en el siguiente lanzamiento de *TI*.

Detalles de Implementación del Ambient Occlusion

Este renderizador también considera dos niveles de recursión: *rayos primarios* y de *sombra*. También está compuesto por tres kernels (figura 3.12 a la derecha), similares a los de PT: generador de rayos (RG), recorrido (TI) y shading (SH). Para completar la imagen final solo es necesario trazar múltiples iteraciones de los rayos de sombra, mientras que los rayos primarios se trazan una única vez.

El kernel RG genera un rayo por píxel, por lo que el número de rayos primarios lanzados en cada ejecución es de 1MRay. El kernel TI tiene dos configuraciones. En la primera, se encuentra el punto de intersección más cercano de cada rayo, lo que es adecuado para rayos primarios. En la segunda, el recorrido termina tan pronto como se encuentra un punto de intersección, lo que corresponde a los rayos de sombra. El kernel SH genera seis rayos de sombra por cada punto de intersección previamente encontrado en TI. Así, son trazados lotes de 6MRays en cada iteración. De manera idéntica a PT, los rayos seleccionan el KD-Tree que deben recorrer en los kernels RG y SH.

Disposición de los Rayos

Los rayos primarios se guardan en el array de rayos según el código de Morton de los píxeles de la imagen. De esta forma, aumenta la probabilidad de que rayos contiguos en el array elijan el mismo KD-Tree. Sin embargo, los rayos secundarios se generan aleatoriamente, por lo que, a diferencia de los primarios, es bastante probable que rayos contiguos elijan KD-Trees diferentes. Pensamos que este hecho da como resultado un aumento de los fallos de caché nada más comenzar el recorrido en el kernel TI, ya que las raíces de los KD-Trees se encuentran muy alejadas en el array *nodes*. Experimentalmente se comprueba que los rayos necesitan de media menos pasos de recorrido con respecto a la SAH original, donde un paso de recorrido es una intersección rayo-plano o una rayo-triángulo. Sin embargo, aunque el número de pasos de recorrido es menor, consumen un tiempo mayor.

Para resolver este inconveniente, se ha añadido un nuevo kernel *Sort* justo antes del kernel TI para los rayos secundarios o de sombra. Este kernel reordena los rayos en función del índice del KD-Tree seleccionado. Para implementar esta ordenación en CUDA se ha usado la primitiva radixsort incluida en CUDPP [HOS+10]. Ya que solo tres valores son requeridos para indicar uno de los tres KD-Trees, la ordenación se lleva a cabo con solo dos bits.

3.9.7. Resultados

La implementación de nuestro sistema se ha realizado sobre una NVidia GeForce 285 GTX con 1GB de memoria RAM sobre las escenas previamente mencionadas (figura 3.11). En las tablas 3.4 y 3.5 se muestran algunos resultados de cada escena para Path Tracing (PT) y Ambient Occlusion (AO), respectivamente. Las tablas están divididas en dos partes. En las dos columnas "SAH" se muestran las estadísticas de un KD-Tree construido con la SAH clásica. En la tres columnas "SAH especializada" se muestran las estadísticas del Multi-KD-Tree que obtiene un mayor rendimiento. El resto de resultados se pueden consultar en el artículo de Torres et al. [TMGA12].

La columna "Pasos" hace referencia al número de *pasos de recorrido* por rayo en media. La columna "MRays/s" mide el rendimiento de los kernels *Sort* y TI en millones de rayos por segundo. Solo se han mostrado datos del kernel TI y del kernel *Sort* ya que estos son los kernels que más tiempo consumen. Concretamente, TI toma entre el 75 %-83 % del tiempo total de renderizado. Las escenas se han evaluado situando varias cámaras en diferentes posiciones y realizando varias ejecuciones con cada una. La columna "Mejor Heurística" indica la heurística especializada con la que fue construido el Multi-KD-Tree que exhibe un rendimiento mayor. En las columnas "Pasos" y

	Rayos Primarios				
	SAH		SAH especializada		
Escena	Pasos	MRays/s	Mejor Heurística	Pasos	MRays/s
Bunny	34.04	141.12	SPHERE-ORTH	30.27(11.08)	147.45(4.29)
FairyForest	48.82	101.70	CUBE-ORTH	45.70(6.38)	106.04(4.09)
ConfRoom	38.46	149.19	SPHERE-OBLI	34.78(9.56)	157.52(5.29)
Sponza	37.66	171.75	SPHERE-ORTH	34.52(8.33)	178.78(3.93)
Sibenik	45.03	143.31	CUBE-OBLI	38.67(14.11)	156.83(8.61)
	Rayos Secundarios				
	SAH		SAH especializada		
Escena	Pasos	MRays/s	Mejor Heurística	Pasos	MRays/s
Bunny	32.09	36.29	SPHERE-OBLI	30.88(3.78)	37.33(2.78)
FairyForest	51.51	19.36	CUBE-OBLI	49.11(4.64)	20.33(4.75)
ConfRoom	39.83	26.21	CUBE-ORTH	38.81(2.55)	27.57(4.93)
Sponza	41.17	26.14	CUBE-OBLI	39.50(4.06)	28.11(7.00)
Sibenik	48.01	19.66	CUBE-OBLI	45.78(4.63)	21.41(8.14)

Tabla 3.4: Mejores resultados para Path Tracing.

	Rayos Primarios				
	SAH		SAH especializada		
Escena	Pasos	MRays/s	Mejor Heurística	Pasos	MRays/s
Bunny	34.23	78.72	SPHERE-ORTH	30.46(10.99)	81.29(3.16)
FairyForest	49.03	63.44	SPHERE-ORTH	45.60(6.99)	65.09(2.53)
ConfRoom	38.55	87.87	SPHERE-OBLI	34.88(9.52)	94.45(6.96)
Sponza	37.73	112.70	SPHERE-ORTH	34.59(8.31)	117.33(3.94)
Sibenik	47.31	79.41	CUBE-OBLI	40.80(13.75)	84.55(6.07)
	Rayos de Sombra				
	SAH		SAH especializada		
Escena	Pasos	MRays/s	Mejor Heurística	Pasos	MRays/s
Bunny	28.91	46.89	SPHERE-OBLI	28.07(2.90)	46.59(-0.63)
FairyForest	42.58	30.80	CUBE-ORTH	40.62(4.59)	31.46(2.10)
ConfRoom	31.28	52.80	CUBE-ORTH	30.77(1.63)	52.98(0.32)
Sponza	34.35	47.02	SPHERE-OBLI	32.96(4.03)	49.03(4.09)
Sibenik	39.55	37.07	CUBE-ORTH	37.94(4.06)	38.79(4.42)

Tabla 3.5: Mejores resultados para Ambient Occlusion.

"MRays/s" se muestra, entre paréntesis, el porcentaje de ahorro en pasos de traversal y de mejora en rendimiento con respecto a la SAH.

Como se puede ver en la columna "Pasos", los rayos requieren menos pasos de recorrido para alcanzar su intersección más cercana con el uso de Multi-KD-Trees, obteniéndose un ahorro de hasta un 14.11 %. En la columna "MRays/s" también se aprecia un aumento en la velocidad de recorrido con respecto a SAH, de hasta 8.61 % para rayos primarios y 8.14 % para rayos secundarios. Sin embargo, este beneficio no es tan grande como en la columna "Pasos" debido a las dependencias que existen entre los hilos, en forma de fallos de caché y divergencias en ejecución.

También hemos probado experimentalmente el comportamiento de las heurísticas COS-ORTH Y COS-OBLI. En la figura 3.13 se muestran los resultados de COS-ORTH para la escena

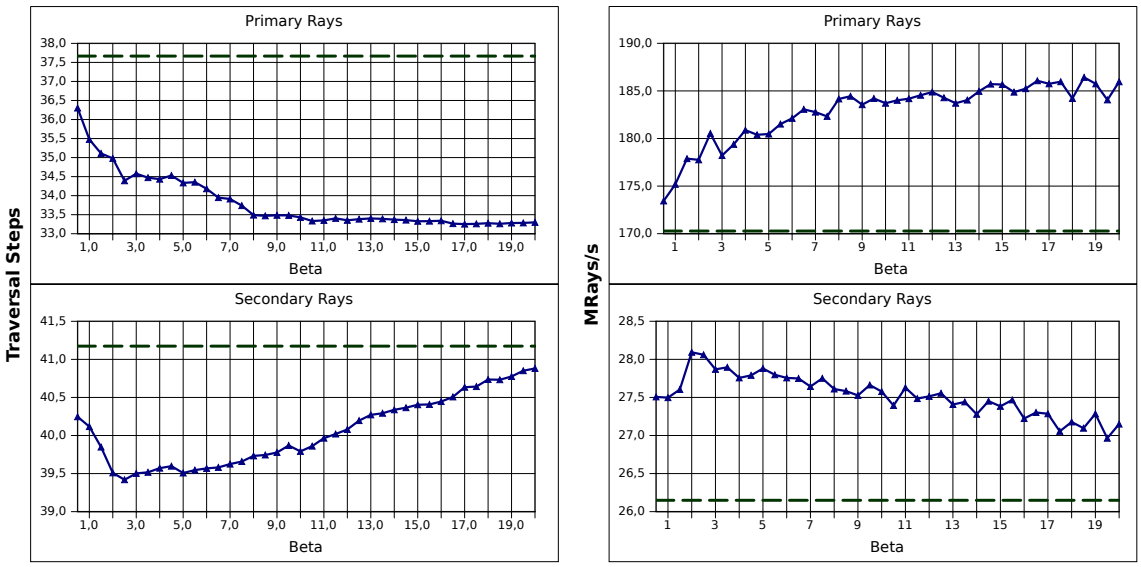


Figura 3.13: Resultados de la heurística COS-ORTH (en azul) para la escena Sponza, renderizada con Path Tracing, en comparación con la misma escena usando la SAH (en rojo).

Sponza renderizada con PT. El resto de los resultados para COS-ORTH se pueden consultar nuevamente en Torres et al [TMGA12]. El parámetro β (sección 3.9.3) varía desde 0.5 hasta 20.0, en pasos de 0.5. Para valores de β menores que 3.5, los pasos de recorrido decrecen. Los pesos resultantes de la heurística COS-ORTH para estos valores de β corresponden aproximadamente a los de las heurísticas SPHERE y CUBE. Para valores superiores de β, el comportamiento es dependiente de la escena.

3.9.8. Discusión sobre las Simetrías de las Heurísticas Especializadas

A la hora de diseñar nuevas heurísticas especializadas, la división del espacio de direcciones debe tener en cuenta las *simetrías* que se derivan de los rayos. La primera fuente de simetrías ya se comentó y proviene de asumir que los rayos se comportan como líneas rectas. Concretamente, esta simetría es consecuencia de suponer que los rayos comienzan y terminan fuera de la escena. Matemáticamente, esto se traduce en que las proyecciones ortogonal y oblicua obtienen la misma medida para conjuntos de rayos con dirección opuestas.

La segunda fuente de simetría se debe a que los volúmenes asociados a cada nodo son cajas alineadas con los ejes. Matemáticamente, esto se traduce en la aparición de valores absolutos en las medidas de probabilidad que dan como resultado la misma probabilidad. Por ejemplo, dos conjuntos de rayos cuyas direcciones sean de la forma ω_x y $-\omega_x$ obtienen los mismos pesos para las caras de una caja, por lo que su probabilidad de intersección será la misma. Igualmente, esto ocurre también para las otras dos componentes de las direcciones.

Estas dos fuentes de simetría imponen restricciones a la forma en que se puede dividir el espacio de direcciones. Así, sería posible considerar conjuntos de rayos cuyas direcciones sean disjuntas pero que den como resultado la misma función de aproximación del coste y, por consiguiente, generen la misma estructura de aceleración, lo cual carecería de sentido.

Existen dos maneras de evitar estas simetrías. Una consiste en asociar a cada nodo de la EA un volumen recubridor que no sea una caja alineada con los ejes. En una BVH esto es fácil si se cambia el volumen recubridor asociado explícitamente a cada nodo. En un KD-Tree, hay que

usar planos de división que no estén alineados con los ejes. La otra manera consiste en suponer que los rayos no comienzan y terminan fuera de la escena, sino que se quedan bloqueados en algún punto de su interior. En ambos casos, y al igual que sucede con cualquier método de aproximación para el coste de una EA, es necesario implementar y evaluar el rendimiento de la correspondiente heurística para comprobar su interés práctico.

Capítulo 4

Aprovechamiento del Hardware

4.1. Introducción

La manera más directa de acelerar los algoritmos de ray tracing es aprovechando al máximo el hardware sobre el que se ejecutan. En la literatura, se han empleado diversas técnicas para aprovechar el hardware (sección 4.4) aunque todas ellas pueden clasificarse en cuatro categorías.

Paralelismo. Las técnicas que caen en esta categoría están orientadas al aprovechamiento de todo el paralelismo disponible en el hardware. Este paralelismo puede venir por dos vías. La primera consiste en disponer de varios procesadores, cada uno de los cuales puede ejecutar un flujo de instrucciones diferente con independencia de los demás. A este paralelismo se le conoce como *MIMD* (de *Multiple Instructions, Multiple Data*). Ejemplos de este hardware son los cores de las CPUs y los SMs (de *Streaming Multiprocessor*) de las GPUs. Estos procesadores se pueden encontrar en diferentes chips conectados por una red o dentro de un mismo chip.

La otra forma de paralelismo consiste en arrays de unidades funcionales donde todas ellas ejecutan la misma instrucción al mismo tiempo, aunque generalmente sobre datos diferentes. A este paralelismo se le llama *SIMD* (de *Single Instruction, Multiple Data*)[1]. Ejemplos son las instrucciones SSE (*Streaming SIMD Extensions*) de las CPUs, o los cores de un SM en las GPUs. El número de operaciones que se pueden realizar en paralelo de manera vectorial recibe el nombre de *anchura* SIMD.

En general, aprovechar el paralelismo de las arquitecturas SIMD es más difícil que el de las MIMD. En una SIMD es necesario tener disponible datos que requieran la misma operación para poder ejecutarla en paralelo, mientras que en una MIMD esta restricción no existe. Si el número de datos sobre los que se opera es menor que la anchura SIMD, entonces se dice que se ha producido una *divergencia*, en cuyo caso, algunas unidades funcionales realizan trabajo inútil. Se llama *eficiencia* SIMD a la fracción entre los datos disponibles y la anchura SIMD. Lo deseable es tener una eficiencia tan alta como sea posible, ya que no siempre es posible llegar al 100 %.

División del trabajo. La segunda categoría incluye aquellas técnicas dedicadas a la división del trabajo. Estas técnicas tienen mucha relación con la categoría anterior ya que, para aprovechar el paralelismo del hardware, el trabajo total que tiene que realizar la aplicación se tiene que dividir en tareas más pequeñas que puedan repartirse entre los diferentes procesadores. Los algoritmos de ray tracing son intrínsecamente paralelos si consideramos que cada ruta es independiente de las demás. Aún así, en la generación de rutas existen partes secuenciales que no se pueden paralelizar. Tal es el caso de la generación de rutas en path tracing o en el ray tracing al estilo de Whitted,

[1] Estas operaciones también se suelen llamar operaciones *vectoriales*. Nosotros usaremos indistintamente los términos SIMD y vectorial.

donde el siguiente rayo no se puede obtener hasta que el rayo anterior haya encontrado su punto de intersección más cercano.

Una vez dividido el trabajo en tareas, estas deben ser asignadas a los procesadores o a las unidades funcionales. Hay que tener en cuenta que todas las tareas pueden no requerir el mismo tiempo para completarse. Por tanto, aún en el caso de un reparto uniforme, es decir, cuando todos los procesadores tienen asignados el mismo número de tareas, puede suceder que algunos procesadores terminen su trabajo antes que los demás, quedándose a la espera de que termine el resto de los procesadores.

Algo similar ocurre con el uso de *barreras*. Es posible que algunas tareas dependan de otras, por lo que tienen que procesarse secuencialmente. Las barreras permiten sincronizar el trabajo, evitando *condiciones de carrera* y la obtención de datos incorrectos. Sin embargo, al igual que antes, puede suceder que un procesador esté mucho tiempo ocioso esperando en la barrera a que lleguen los demás.

Sistema de memoria. En esta tercera categoría se encuentran las técnicas destinadas a optimizar el sistema de memoria. Un buen diseño de los algoritmos y de la capa de memoria permite un aumento del número de aciertos de las memorias cachés con la consecuente disminución de las transferencias de memoria. Una manera de mejorar el uso de la jerarquía de memoria es aprovechando la localidad espacial y temporal que presentan determinados algoritmos. La *localidad* de un algoritmo consiste en la probabilidad de que, tras acceder a un dato, se acceda a datos cercanos en memoria (localidad espacial) o que se repita el acceso al mismo dato poco después (localidad temporal). Esto permite mantener algunos datos en una memoria rápida y evitar el trasiego de información con la memoria principal, que suele ser bastante lenta. Ejemplos de estas memorias rápidas en GPU son los registros, la memoria compartida y las cachés, todas ellas memorias on-chip.

Guardar los datos en memoria como una estructura de arrays (*SoA*, de *Structure of Arrays*), en vez de un array de estructuras (*AoS*, de *Array of Structures*), es una manera de disminuir el tráfico con memoria. Un AoS consiste en que cada elemento del array mantiene todos los campos de la estructura, mientras que en una SoA cada campo se guarda de forma consecutiva en un array. Por ejemplo, en GPU el siguiente tipo puede usarse para representar un rayo

```
struct ray_t {
    float4 origin;
    float4 direction;
};
```

El tipo `float4` es una estructura con cuatro reales en coma flotante. Los tres primeros reales de `origin` y `direction` guardan las tres coordenadas del origen y de la dirección del rayo, respectivamente, mientras que los últimos reales solo se usan para el alineamiento de memoria[2]. Un AoS de 100 rayos se corresponde con la siguiente declaración

```
ray_t rays[100];
```

Cuando los 32 hilos de un warp traigan sus 32 rayos consecutivos desde memoria global a registros, ejecutarán el siguiente código

```
float4 o = rays[thid].origin;
float4 d = rays[thid].direction;
```

donde `thid` es el identificador del hilo. Las lecturas de los orígenes de los rayos generan 8 transacciones de 128 bytes cada una desde memoria, ya que estos no se encuentran en posiciones consecutivas. Las lecturas de las direcciones generan también otras 8 transacciones de memoria, resultando

[2]Aunque en este sencillo ejemplo se desperdician 8 bytes de memoria (equivalentes a dos `floats`) por cada rayo, en la práctica esos `floats` se usan para almacenar más información como, por ejemplo, un identificador de BRDF o la semilla para generar posteriores números pseudo-aleatorios.

en un total de 16 transacciones para que los hilos de un warp lean 32 rayos[3].

El número de transacciones se puede reducir a 8 si los rayos se guardan como una SoA, de la siguiente manera

```
struct {
    float4 origins[100];
    float4 directions[100];
} rays;
```

Así, la lectura de un rayo por parte de un hilo se realiza como sigue

```
float4 o = rays.origins[thid];
float4 d = rays.directions[thid];
```

En este caso, solo 8 transacciones de 128 bytes son necesarias para que un warp lea 32 rayos, ya que tanto los orígenes como las direcciones ocupan posiciones consecutivas en memoria.

El mutithreading es también una forma de aprovechar el sistema de memoria. Consiste en asignar a un procesador más hilos de los que puede ejecutar a la vez. Esto permite que, mientras que un hilo espera a que llegue el dato que ha solicitado de memoria, otro pueda ejecutarse. A esta acción se le conoce como *ocultar las latencias* de memoria (sección 2.2.2). El inconveniente que tiene es que se requieren más recursos para mantener simultáneamente el estado de todos los hilos.

Optimización del código. Esta última categoría corresponde a la cuidadosa codificación del programa. Así, si el programa requiere menos operaciones para completar una tarea o consume menos recursos es posible que sea más rápido. Técnicas que mantienen información precalculada, y, por consiguiente, que evitan realizar algunas operaciones extras, caen dentro de esta categoría.

4.2. Coherencia

Para aprovechar los recursos hardware comentados anteriormente, en la literatura se ha usado el concepto de *coherencia* de un grupo de rayos. No existe una definición precisa del concepto de coherencia, ya que cambia en función del artículo o de los autores. Sin embargo, todas las definiciones coinciden en que el objetivo es aprovechar alguna característica común a los rayos de un grupo para beneficiarse de su ejecución en el hardware.

La característica común de los rayos de un conjunto usada con mayor frecuencia consiste en analizar el conjunto de nodos de la EA que estos rayos visitan durante su recorrido. Así, se dice que un grupo de rayos es coherente si sus rayos visitan los mismos nodos de la EA e intersecan con los mismos triángulos durante la mayor parte del recorrido. Se derivan varias ventajas si un grupo de rayos coherentes recorre juntos la EA. Primero, el grupo exhibe localidad ya que el mismo nodo o triángulo es demandado por todos los rayos del grupo, lo que mejora el rendimiento del sistema de memoria. Segundo, las operaciones que realizan los rayos son las mismas, ya que todos intersecan con los mismos objetos, por lo que es posible aplicar operaciones vectoriales para ejecutar estas tareas.

La formación de un grupo de rayos coherentes se puede realizar antes o durante el recorrido. Si se realiza antes, es necesario un algoritmo de agrupamiento, posiblemente heurístico. Este tipo de agrupamiento, que denominamos *agrupamiento a priori* (sección 4.2.1), suele estar basado en la información geométrica de los rayos (orígenes y direcciones), ya que siempre está disponible antes del recorrido. En cambio, si la agrupación se realiza múltiples veces durante el propio recorrido en función del siguiente nodo que visitará cada rayo, entonces resulta posible mantener mayores niveles de coherencia. A este agrupamiento lo llamamos *agrupamiento por comportamiento* (sección 4.2.2). Sin embargo, el agrupamiento por comportamiento presenta una mayor sobrecarga que

[3] Esto es también consecuencia de que las instrucciones de memoria puede leer hasta 16 bytes como máximo (=`float4`), por lo que la lectura de los orígenes y las direcciones de los rayos se tiene que realizar por separado.

el agrupamiento a priori debido a que la operación de agrupamiento se realiza varias veces durante el recorrido.

Por restricciones específicas del hardware, un grupo de rayos puede pasar a tratarse como un *todo* durante el recorrido, convirtiéndose así en la nueva unidad de recorrido. Si esto sucede, a ese grupo de rayos se le denomina *paquete* de rayos. La definición de paquete de rayos difiere en función de los autores y de los trabajos. Nosotros llamaremos paquete de rayos a un grupo de rayos que el algoritmo de recorrido maneja explícitamente. Si el recorrido tiene al paquete como unidad de recorrido, entonces lo llamaremos recorrido basado en paquetes (*packet-based traversal*); si, por el contrario, el recorrido de cada rayo es independiente de los demás, entonces lo llamaremos recorrido por rayo (*single-ray traversal*).

4.2.1. Agrupamiento a Priori

Decimos que un conjunto de rayos son *geométricamente parecidos* si tienen orígenes cercanos y direcciones semejantes. Si esto se cumple, es muy probable que estos rayos recorran los mismos nodos de la EA e intersequen con los mimos triángulos durante gran parte de su recorrido, sobre todo al comienzo. Sin embargo, el concepto de rayos geométricamente "parecidos", así como el de orígenes y direcciones "cercanos", no es lo suficientemente preciso como para agrupar rayos, por lo que es necesario el uso de heurísticas.

Para los rayos primarios, la heurística habitual de agrupamiento consiste en considerar los píxeles sobre los que se han generado. Así, cuanto más cercanos sean estos píxeles, sus direcciones estarán más próximas, y la probabilidad de que visiten los mismos nodos y triángulos de la EA será mayor.

Los rayos secundarios se pueden agrupar conservando la misma agrupación que tenían los rayos primarios que los generaron. Sin embargo, la coherencia de un conjunto de rayos disminuye en cada rebote. Dos causas influyen en este hecho. La primera es debida a la propia geometría de la escena. Así, aunque dos rayos primarios hayan sido generados sobre píxeles contiguos —esto es, comparten el mismo origen y sus direcciones son muy parecidas— es posible que sus puntos de intersección estén muy alejados, lo que suele ocurrir en los bordes de los objetos. La segunda causa de disminución de coherencia en cada rebote depende del tipo de algoritmo de renderizado. Si el algoritmo es un ray tracing al estilo de Whitted (RTW) o distribuido (RTD), entonces la disminución en la coherencia es menor que en otros algoritmos, por ejemplo, path tracing (PT). Esto se debe a que la dirección del siguiente rayo de la ruta solo puede ser tomada sobre un conjunto pequeño de posibilidades (Boulos et al. [BEL+07]).

La disminución de la coherencia dentro de un grupo de rayos no impide que pueda aparecer nueva coherencia entre rayos de diferentes grupos. Para aprovecharla es necesario un reagrupamiento de los rayos, lo que supone una carga extra en el recorrido. Un agrupamiento eficaz y rápido de estos rayos es todavía un tema activo de investigación.

El agrupamiento a priori no garantiza que los rayos de un grupo visiten exactamente los mismos nodos y triángulos durante el recorrido, por lo que las divergencias tienen que ser tratadas explícitamente. Un buen agrupamiento significa que esta propiedad se cumple la mayor parte del tiempo, aumentando con ello el rendimiento global del recorrido. Por el contrario, un mal agrupamiento requiere más tiempo para conseguir que cada rayo encuentre su punto de intersección más cercano.

4.2.2. Agrupamiento por Comportamiento

En el agrupamiento por comportamiento se tienen en cuenta directamente los nodos y los triángulos que cada rayo visita. De esta manera, aquellos rayos cuyo siguiente nodo o triángulo por visitar sea el mismo serán factibles de ser agrupados. Con un mayor nivel de detalle, se pueden

tener en cuenta incluso las operaciones que cada rayo pretende realizar en vez de solo su siguiente nodo por explorar.

Esta forma de agrupamiento requiere que los rayos sean agrupados en cada paso del recorrido, lo que aumenta significativamente la sobrecarga. Sin embargo, la coherencia que se obtiene es perfecta porque los grupos van a estar formados solo por aquellos rayos que visitarán el mismo nodo o triángulo. Una forma de disminuir la sobrecarga que genera este agrupamiento es realizándolo solo cada cierto número de pasos del recorrido, o solo en ciertos niveles de la EA. Esta aproximación se puede entender entonces como una mezcla entre los dos tipos de agrupamiento, ya que los rayos se agrupan con más frecuencia que en el agrupamiento a priori, que solo ocurre antes del recorrido, pero no tan a menudo como en el agrupamiento por comportamiento. En este caso, también habría que manejar las incoherencias debidas a que algunos rayos del grupo recorran diferentes nodos.

El beneficio en el rendimiento de todas las técnicas de agrupación de rayos en grupos coherentes debe ser probado experimentalmente para comprobar su utilidad práctica. En este momento sigue siendo una cuestión abierta determinar qué técnica de agrupación resulta más beneficiosa.

4.3. Recorrido en GPU

Aila y Laine [AL09] presentan un análisis de diferentes implementaciones de algoritmos de recorrido sobre la GPU GeForce GTX 285 (cap. 1.3), que luego fue extendido a arquitecturas Fermi y Kepler por Aila et al. [ALK12]. La EA usada fue una BVH, a la que se le había aplicado *early split clipping* [EG07] para reducir el impacto de los triángulos grandes. Los algoritmos usados se describen a continuación.

(1) Packet. Es el recorrido de una BVH con paquetes usado por Günther et al. [GPSS07]. Cada paquete está formado por los 32 hilos de un warp (32 rayos), que comparten una pila de recorrido común, implementada en memoria compartida. Todos los accesos a memoria que se producen durante el recorrido están siempre fusionados. Sin embargo, se da el hecho de que el número de nodos visitados es mayor que si cada rayo recorriera individualmente la EA.

(2) While-while. Es un algoritmo de recorrido por rayo[4], y no por paquete. Cada hilo mantiene su propia pila de recorrido, implementada en memoria local[5]. Este algoritmo usa dos bucles `while` secuenciados en la implementación del recorrido, de ahí su nombre. En el primer bucle, el rayo recorre los nodos de la BVH hasta llegar a una hoja. Una vez allí, el segundo bucle se encarga de realizar la intersección del rayo con todos los triángulos de la hoja. Cuando ha terminado el segundo bucle, se vuelven a ejecutar de nuevo los dos bucles, hasta que la pila de recorrido queda vacía.

(3) If-if. Es una versión del recorrido parecida a *while-while*, pero usando dos sentencias `if` en vez de los dos bucles `while`. Cada `if` se encarga, respectivamente, de comprobar si el nodo es interno u hoja, y de realizar la acción correspondiente.

(4) Persistent packet y persistent while-while. Son versiones de los algoritmos *packet* y *while-while* anteriores pero que usan hilos persistentes (sección 4.3.1). Las GPUs de cap. 1.3 reparten de manera uniforme los bloques de hilos entre sus multiprocesadores. Los hilos persistentes son una forma de evitar ese reparto uniforme y realizar uno dirigido por la demanda.

(5) Speculative persistent while-while. Este es el algoritmo *persistent while-while* al que se le ha añadido *especulación* sobre los nodos. Así, un rayo realizará la intersección con la caja de un nodo interno, aunque su siguiente nodo sea hoja, si el resto de hilos de su warp también lo hacen. Esto evita que el hilo se quede esperando a que el resto de hilos termine y posiblemente

[4] Creemos que Zhou et al. [ZHWG08] fueron los primeros en implementar un algoritmo de recorrido por rayo en GPU. Suponemos que los buenos resultados que obtuvieron inspiraron este artículo de Aila y Laine.

[5] La memoria local no existe como un hardware diferente. En una GPU con cap. 1.3 (la que se usa en el artículo) la memoria local se implementa como memoria global. A partir de cap. 2.0, la memoria local reside en la caché L1.

adelante trabajo, con el coste extra de traer un nodo de más desde memoria.

Los autores compararon el rendimiento real de estos algoritmos de recorrido con su rendimiento en simulación. Durante la ejecución simulada no se tuvieron en cuenta las posibles paradas producidas por las unidades funcionales o por las latencias de memoria, por ello, su estudio solo revela una cota máxima en tiempo de ejecución. No obstante, la diferencia entre el rendimiento medido y el simulado dio suficiente información a los autores como para llegar a las siguientes conclusiones:

(a) Persistente *vs.* no persistente. Las latencias de memoria son un factor limitador, pero no el único. En concreto, los autores observaron que el tiempo de ejecución puede variar mucho de unos rayos a otros, ocasionando variaciones en los tiempos de ejecución entre los warps. Por tanto, un reparto uniforme de la carga de trabajo (en este caso, las tareas son lotes de 32 rayos) no es la mejor opción. El uso de hilos persistentes es la manera de realizar un reparto dirigido por la demanda, lo que explica que estos algoritmos tengan un rendimiento mejor que sus contrapartidas no persistentes.

(b) Single *vs.* packet. Un recorrido por rayo realiza menos peticiones a memoria, aunque las peticiones agregadas de todos los hilos de un warp suelen ser más incoherentes. Esto implica la producción de más transacciones de memoria porque los accesos son más difíciles de fusionar. Sin embargo, en un recorrido basado en paquetes el número de peticiones a memoria es mayor, pero estas son más coherentes, produciéndose menos transacciones. Experimentalmente se ha comprobado que el recorrido por rayo tiene un mejor rendimiento que el basado en paquetes.

(c) Speculative *vs.* non-speculative El algoritmo especulativo tiene una mayor efectividad SIMD ya que intenta no dejar ningún canal SIMD sin usar. Sin embargo, aumentar la eficiencia SIMD requiere realizar alguna consulta de nodos extra. El rendimiento del especulativo es mayor para los rayos primarios, aunque no mucho mayor. Para los secundarios, llega a ser menor en algunas escenas.

4.3.1. Hilos Persistentes

En el algoritmo de los hilos persistentes, cada rayo es procesado únicamente por un hilo. Sin embargo, cada hilo procesa secuencialmente tantos rayos como le sea posible, hasta que se agote el lote de rayos. La idea detrás de este esquema es que la carga de los warps esté dirigida por la demanda. Al ejecutar el algoritmo, se lanza el máximo número de hilos que puedan residir en la GPU. Estos hilos, llamados *hilos persistentes*, solo terminan cuando se han procesado todos los datos. El número concreto de hilos depende, sobre todo, del número de registros y de la memoria compartida disponible en la GPU. El esquema de un algoritmo de recorrido con hilos persistentes se muestra en la figura 4.1.

La variable `globalNextRay` (línea 2) indica el número de rayos ya procesados. El primer hilo de cada warp (línea 19) se encarga de reservar un lote de 32 rayos[6], por eso suma 32 a `globalNextRay` (línea 20). Se requiere que la suma sea atómica para que no exista interferencia con otros hilos en ejecución. Una vez reservados los rayos, pueden traerse desde memoria (línea 31) ya que se tiene la seguridad de que ningún otro hilo los tiene reservados. Finalmente, cada rayo se envía a la función concreta de recorrido (línea 34) para que encuentre su punto de intersección más cercano. Una vez terminado de procesar los 32 rayos, el warp vuelve a intentar reservar otros 32 rayos (línea 9). El hilo termina cuando se le asigna un índice de rayo que se encuentra fuera del array de rayos (líneas 27–28).

[6] En la práctica, el lote que se reserva es un múltiplo de 32, lo que evita muchas llamadas a la función `atomicAdd` para no sobrecargar el sistema de memoria. Nosotros hemos usado lotes de $3 \cdot 32$ rayos.

```
// Indice global sobre el array de rayos.
__device__ int globalNextRay = 0;

// Kernel de recorrido.
__global__ void traversal(...) {

  // Bucle de los hilos persistentes. Solo se sale de este bucle cuando
  // no quedan más rayos por procesar.
  while(true){

    // Indice del hilo dentro de un warp.
    int lane = thid % 32;

    // Comienzo del conjunto de 32 rayos reservados por el warp.
    __shared__ int localNextRay;

    // El primer hilo del warp se encarga de reservar 32 rayos
    // sumando atómicamente 32 al contador global.
    if (lane == 0){
      localNextRay = atomicAdd(globalNextRay, 32);
    }

    // Indice global del rayo del hilo.
    int myRayId = localNextRay + lane;

    // Si el índice apunta fuera del array, entonces el hilo termina.
    if (myRayId >= raysSize)
      return;

    // Cada hilo trae su rayo.
    ray_t ray = raysPool[myRayId];

    // Rutina de recorrido concreta.
    trace(ray);
  } // while(true)
} // traversal()
```

Figura 4.1: Esquema de los algoritmos de recorrido que usan hilos persistentes.

4.3.2. Intersección Rayo-Triángulo en GPU

Dos algoritmos de intersección rayo-triángulo se han usado en esta tesis: el de Möller y Trumbore [MT97] y el de S. Woop [Woo04]. En el algoritmo de Möller y Trumbore se realiza la intersección directa de un rayo con el triángulo representado por sus tres vértices. Los autores optimizan la implementación de este algoritmo para evitar que ciertas operaciones se realicen más de una vez. En el algoritmo de S. Woop, cada triángulo se transforma en otro coplanario al plano XY, antes de realizar la intersección.

El algoritmo de Möller y Trumbore consume menos memoria ya que solo tiene que guardar $3 \cdot 3$ reales por triángulo, que corresponden a las tres coordenadas de sus tres vértices. El algoritmo de S. Woop tiene que mantener la matriz de transformación 4×4, que consume $3 \cdot 4$ reales (la última fila de la matriz es constante y no es necesario guardarla). Sin embargo, el código del algoritmo de S. Woop consume menos registros en GPU, por lo que la ocupación de su kernel es mayor que el de Möller y Trumbore, lo que supone un mejor rendimiento en GPU. Por esto motivo, es el algoritmo de intersección rayo-triángulo que hemos usado en GPU desde que tuvimos noticia de él en el trabajo de Aila y Laine [AL09].

Intersección de Möller y Trumbore

Los puntos pertenecientes a un rayo son aquellos de la forma $o + t \cdot d$, donde $t > 0$. Los puntos interiores a un triángulo con vértices a, b y c tienen la forma $(1 - u - v) \cdot a + u \cdot b + v \cdot c$, donde $u \geq 0$, $v \geq 0$ y $u + v \leq 1$, es decir, (u, v) son las *coordenadas baricéntricas* de los puntos del triángulo. Por tanto, el punto de intersección rayo-triángulo es aquel que cumple la ecuación

$$o + t_{hit} \cdot d \quad = \quad (1 - u_{hit} - v_{hit}) \cdot a \; + \; u_{hit} \cdot b \; + \; v_{hit} \cdot c \tag{4.1}$$

más las restricciones para t_{hit}, u_{hit} y v_{hit}

$$t_{hit} > 0, \quad u_{hit} \geq 0, \quad v_{hit} \geq 0 \quad \text{y} \quad u_{hit} + v_{hit} \leq 1 \tag{4.2}$$

Poniendo el sistema de ecuaciones 4.1 en forma matricial, tenemos

$$M \cdot \begin{bmatrix} t_{hit} \\ u_{hit} \\ v_{hit} \end{bmatrix} = o - a, \tag{4.3}$$

donde M es una matriz 3×3 formada por los tres vectores columna $[-d \quad b-a \quad c-a]$. Resolviendo el sistema de ecuaciones 4.3, tenemos

$$\begin{bmatrix} t_{hit} \\ u_{hit} \\ v_{hit} \end{bmatrix} = M^{-1} \cdot (o - a) = \frac{1}{det([-d \quad E_1 \quad E_2])} \begin{bmatrix} det([T \quad E_1 \quad E_2]) \\ det([-d \quad T \quad E_2]) \\ det([-d \quad E_1 \quad T]) \end{bmatrix}$$

donde $E_1 = b - a$, $E_2 = c - a$, y $T = o - a$. Sabiendo que el determinante $det([A \quad B \quad C]) = (A \times B) \cdot C$ es el producto mixto de tres vectores, el sistema finalmente queda como sigue

$$\begin{bmatrix} t_{hit} \\ u_{hit} \\ v_{hit} \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} (Q \cdot E_2) \\ (P \cdot T) \\ (Q \cdot d) \end{bmatrix} \tag{4.4}$$

donde $P = (d \times E_2)$, $Q = (T \times E_1)$. Una vez resuelto el sistema de ecuaciones 4.4, hay que comprobar las restricciones 4.2. En caso de que no se cumpliera alguna, no existiría intersección.

Hay un caso en el que no se puede resolver el sistema. Ese caso aparece cuando $P \cdot E_1 = 0$ y ocurre cuando el rayo es paralelo al triángulo. En la práctica, este caso se comprueba antes de resolver el sistema y se considera que no existe intersección entre el rayo y el triángulo.

Intersección de Woop

La intersección rayo-triángulo que usa el algoritmo de Aila y Laine [AL09] se puede encontrar en el trabajo de Schmittler et al. [SWW$^+$04], aunque una explicación más detallada se encuentra en el capítulo 5 de la tesis de máster de S. Woop [Woo04].

Una transformación afín T está especificada por una matriz M de dimensión 3×3, más un vector desplazamiento N de dimensión 3×1. La aplicación de T sobre un punto p se define como $T(p) = M \cdot p + N$, mientras que sobre un vector v se define como $T(v) = M \cdot v$.

Dado un triángulo Δ con vértices en los puntos a, b y c y normal n, buscamos construir la transformación afín T, tal que

$$T(a) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad T(b) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad T(c) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{y} \quad T(n) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Esta transformación T lleva el triángulo Δ a otro triángulo cuyos vértices se encuentran en $(1,0,0)$, $(0,1,0)$ y $(0,0,0)$, y tiene por normal $(0,0,1)$. A este triángulo coplanario con el plano XY le llamamos Δ'. Para obtener T, primero construimos la aplicación afín inversa $T^{-1}(x) = M' \cdot x + N'$, que se obtiene resolviendo el siguiente sistema

$$T^{-1}(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}) = a, \quad T^{-1}(\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}) = b, \quad T^{-1}(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}) = c \quad \text{y} \quad T^{-1}(\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}) = n.$$

Por tanto,

$$M' = \begin{bmatrix} a_x - c_x & b_x - c_x & n_x \\ a_y - c_y & b_y - c_y & n_y \\ a_z - c_z & b_z - c_z & n_z \end{bmatrix} \quad \text{y} \quad N' = \begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix}$$

La transformación T se obtiene a partir de la inversa de T^{-1} como sigue

$$M = (M')^{-1} = \frac{1}{det(M')} tras(adj(M')), \quad N = -M \cdot N'$$

donde *det*, *tras* y *adj* son el determinante, la matriz traspuesta y la adjunta, respectivamente.

Para realizar la intersección de un rayo con un triángulo, primero hay que transformar el rayo al sistema de coordenadas del triángulo Δ'. Esto se consigue aplicando la transformación T al rayo, y posteriormente realizando la intersección rayo-triángulo. Aplicar la transformación al rayo r resulta en el rayo $r' = T(r)$, con

$$o' = T(o) = M \cdot o + N$$
$$d' = T(d) = M \cdot d$$

donde o y d son el origen y dirección de r, y o' y d' son el origen y dirección de r'.

Ahora, la intersección de r' con Δ' es muy fácil de calcular debido a la posición de los vértices del triángulo. El tiempo t_{hit} del punto de intersección rayo-triángulo y sus coordenadas baricéntricas (u_{hit}, v_{hit}) son las mismas para la intersección del rayo r con el triángulo Δ que para las de r' con Δ'. Sus valores se calculan como

$$t_{hit} = -\frac{o'_z}{d'_z}, \quad u_{hit} = t_{hit} \cdot d'_x + o'_x \quad \text{y} \quad v_{hit} = t_{hit} \cdot d'_y + o'_y$$

Para que exista intersección con el triángulo, se deben cumplir las restricciones de 4.2. En la figura 4.2 se muestra el código en CUDA para computar la intersección. Ya que los vértices a, b y c del triángulo y su normal n no se usan durante el recorrido, solo es necesario mantener las matrices M y N de la transformación T. Dichas matrices se concatenan en una matriz 3×4 guardada por filas, que se traen durante la intersección (líneas 2, 14 y 26).

Si el rayo es paralelo al triángulo, entonces $d'_z = 0$ (puesto que las transformaciones afines preservan rectas paralelas). Por tanto, al calcular t_{hit} se puede obtener $\pm\infty$ si $o'_z \neq 0$, o *not-a-number* (*NaN*) si $o'_z = 0$. En cualquiera de los dos casos, no se cumple la condición `thit > 0` de la línea 11 del algoritmo, y no existe intersección del rayo con el triángulo, no siendo necesario tratar explícitamente estos casos.

4.3.3. Intersección Rayo-Caja en GPU

El algoritmo de intersección rayo-caja que se describe en esta sección es una versión del algoritmo que se encuentra en el artículo de Williams et al. [WBMS05], a su vez inspirado en

```
1  // Traemos la fila 2 de la transformación del triangulo.
2  float4 row = triangleTrans[idTri].fila2;
3
4  // Componente 'z' del origen y de la dirección del rayo transformado.
5  float o = row.x * ori.x + row.y * ori.y + row.z * ori.z + row.w;
6  float d = row.x * dir.x + row.y * dir.y + row.z * dir.z;
7
8  // Tiempo de intersección.
9  float thit = - o / d;
10
11 if ((thit > 0.0f) && (thit < tmin)) {
12
13     // Traemos la fila 0 de la transformación.
14     row = triangleTrans[idTri].fila0;
15
16     // Componente 'x' del origen y de la dirección del rayo transformado.
17     o = row.x * ori.x + row.y * ori.y + row.z * ori.z  + row.w;
18     d = row.x * dir.x + row.y * dir.y + row.z * dir.z;
19
20     // Coordenada baricéntrica 'u'.
21     float u = o + thit * d;
22
23     if (u >= 0.0f) {
24
25         // Traemos la fila 1 de la transformación.
26         row = triangleTrans[idTri].fila1;
27
28         // Componente 'y' del origen y de la dirección del rayo transformado.
29         o = row.x * ori.x + row.y * ori.y + row.z * ori.z + row.w;
30         d = row.x * dir.x + row.y * dir.y + row.z * dir.z;
31
32         // Coordenada baricéntrica 'v'.
33         float v = o + thit * d;
34
35         if ((v >= 0.0f) && ((u+v) <= 1.0f)) {
36             // Actualizamos la intersección menor hasta ahora.
37             tmin = thit;
38             // Guardamos el punto de intersección en memoria.
39             ...
40         }
41     }
42 }
```

Figura 4.2: Algoritmo de intersección de Woop entre un rayo y un triángulo.

el artículo de B. Smits [Smi98]. Una explicación alternativa se puede encontrar también en el libro de Shirley y Morley [SM03]. El código que se muestra en la figura 4.3 no es exactamente el código implementado, sino una simplificación para facilitar su explicación. El código usado se puede encontrar en Karras et al. [KAL12].

En las líneas 9–14 del algoritmo, se obtienen los tiempos de intersección del rayo con cada par de planos paralelos de la caja alineada con los ejes (variables `tx0`, `tx1`, `ty0`, `ty1`, `tz0` y `tz1`). Debido a que las componentes de la dirección del rayo pueden ser negativas, los tiempos de intersección anteriores no están ordenados. Por eso, se calcula el mínimo y el máximo de cada pareja de tiempos para obtener los tiempos de entrada y salida de cada par de planos (líneas 17–18).

Al ser un objeto convexo, el intervalo de intersección de un rayo con una caja es la intersección de los intervalos correspondientes a las tres parejas de planos. Por ello se procede calculando el máximo de los tiempos de entrada y el mínimo de los tiempos de salida (líneas 21–22).

```
1  float3 inv_dir;
2  float tx0, tx1, ty0, ty1, tz0, tz1;
3  float3 tentry_planes, texit_planes;
4
5  // Inversa de la dirección del rayo.
6  inv_dir.x = 1.0f/d.x; inv_dir.y = 1.0f/d.y; inv_dir.z = 1.0f/d.z;
7
8  // Intersecciones del rayo con los seis planos.
9  tx0 = (box.min.x - o.x) * inv_dir.x;
10 tx1 = (box.max.x - o.x) * inv_dir.x;
11 ty0 = (box.min.y - o.y) * inv_dir.y;
12 ty1 = (box.max.y - o.y) * inv_dir.y;
13 tz0 = (box.min.z - o.z) * inv_dir.z;
14 tz1 = (box.max.z - o.z) * inv_dir.z;
15
16 // Puntos de entrada y salida de cada plano.
17 tentry_planes = make_float3(min(tx0, tx1), min(ty0, ty1), min(tz0, tz1));
18 texit_planes  = make_float3(max(tx0, tx1), max(ty0, ty1), max(tz0, tz1));
19
20 // Puntos de entrada y salida de la caja.
21 tentry = max(tentry_planes.x, tentry_planes.y, tentry_planes.z);
22 texit  = min(texit_planes.x,  texit_planes.y,  texit_planes.z);
23
24 // Comprobación final.
25 bool result = (tentry <= texit) && (texit >= 0.0f) && (tentry < tmin);
```

Figura 4.3: Algoritmo de intersección entre un rayo y una caja.

En la línea 25 se comprueba si ha habido intersección real con la caja. La primera condición (`tentry <= texit`) indica que el intervalo no es vacío, lo que significa que sí existe intersección entre la recta del rayo y la caja. La segunda condición (`texit >= 0.0f`) indica que la caja no está detrás del rayo o, equivalentemente, que el origen del rayo está antes o dentro de la caja. La tercera condición (`tentry < tmin`) sirve para descartar la caja en caso de que esté más allá del tiempo mínimo encontrado hasta el momento, llamado early culling (sección 3.2), lo que indicaría que en esa caja no puede existir un triángulo con intersección anterior a t_{min}.

4.3.4. Código de Morton

El código de Morton o *z-orden* es una forma de ordenar los elementos de un array bidimensional. Sean (i, j) los índices de la fila y la columna, respectivamente, de un elemento del array. Cada una de estas coordenadas puede ser representada en base 2 como $i = a_n \dots a_1$ y $j = b_n \dots b_1$, donde n es el tamaño de cada dimensión del array, y $a_1, \dots, a_n, b_1, \dots, b_n \in \{0, 1\}$ son los dígitos binarios de cada coordenada. El código de Morton m de estas dos coordenadas es el entero resultante de entrelazar las dos representaciones binarias anteriores, es decir, $m = b_n a_n \dots b_1 a_1$. En la figura 4.4 se muestra un ejemplo para $n = 3$.

El código de Morton se usa para guardar un array bidimensional en memoria lineal y mantener, con bastante eficacia, la proximidad de sus elementos. El código de Morton de cada celda, obtenido a partir de sus dos coordenadas, se usa como índice sobre el array de destino en memoria lineal. Así, si la matriz de partida se divide en submatrices cuadradas cuyo número de filas y columnas es potencia de dos, entonces los elementos de cada submatriz quedan contiguos en el array de destino. Esta propiedad también se cumple si las submatrices no son cuadradas y el número de columnas es el doble que el de filas.

En los algoritmos de ray tracing, este código se usa para ordenar los rayos primarios en

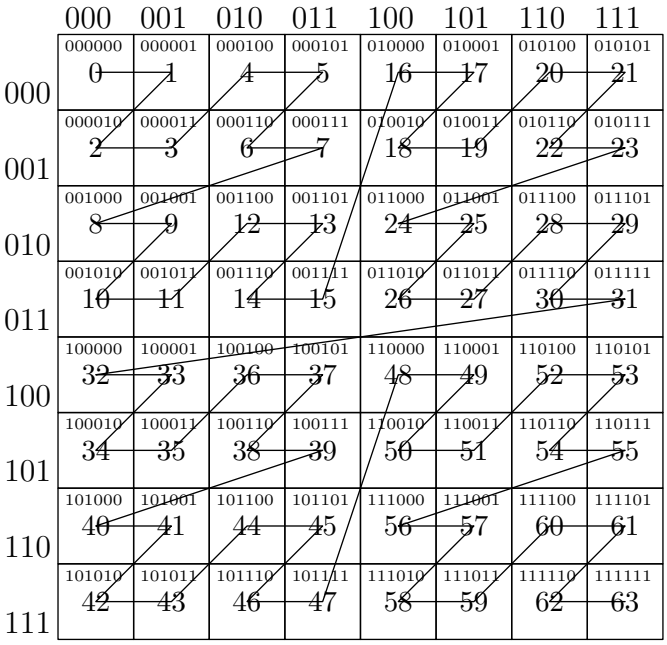


Figura 4.4: Ejemplo de código de Morton para un array de 8×8 elementos. En la parte superior y en el centro de cada celda se muestra su código de Morton en binario y en decimal, respectivamente. La línea zigzagueante une las celdas siguiendo sus códigos de Morton en orden creciente.

función de a qué píxel pertenecen, aprovechando, de esta manera, la coherencia de los mismos. Primeramente, se lanzan tantos hilos como rayos primarios se desean generar. El identificador de cada hilo será el código de Morton del píxel sobre el que se va a generar el rayo primario. El algoritmo usado para obtener las dos coordenadas del píxel a partir del código de Morton se muestra en la figura 4.5. Este algoritmo se encarga de extraer cada bit de cada coordenada mediante desplazamientos y operaciones bit a bit.

```
1  // morton: código de Morton
2  // n: número de bits en cada coordenada
3  // x, y: coordenadas del píxel
4  void from_Morton_to_2D(int morton, int n, int& x, int& y) {
5      x = 0;
6      y = 0;
7      int mask = 1;
8      for(int i = 0; i < n; i++) {
9          x = x | (mask & morton);
10         morton = morton >> 1;
11         y = y | (mask & morton);
12         mask = mask << 1;
13     }
14 }
```

Figura 4.5: Algoritmo que devuelve las dos coordenadas de un píxel a partir de su código de Morton.

4.4. Trabajos Relacionados

Se han realizado muchos trabajos para aprovechar el hardware sobre el que se ejecutan los algoritmos de ray tracing, llegando a ofrecer interacción en tiempo real en determinadas situaciones. Los primeros trabajos se realizaron en CPU, antes de que CUDA estuviera disponible. En este sentido, los trabajos de I. Wald [Wal04] y el uso de paquetes de rayos (Wald et al. [WBWS01]) marcaron un punto de inflexión en la implementación de sistemas de ray tracing eficientes en CPU. Posteriormente, algunos de esos conceptos fueron exportados desde CPU a GPU hasta que se presentó el algoritmo de recorrido más usado en GPU, debido a Aila y Laine [AL09].

4.4.1. Trabajos Relacionados en CPU

Agrupamiento a priori

J. Amanatides [Ama84] presenta un trabajo en el que los rayos se sustituyen por conos durante el renderizado de la escena. Así, por cada píxel se traza un cono, situando su vértice en el origen de la cámara de manera que el cono cubra el área correspondiente a su píxel. Durante el recorrido, por cada cono se mantiene una lista con los objetos que interseca junto con el porcentaje de oclusión de cada uno. Esto permite implementar antialiasing, sombras suaves y materiales glossy, imposibles de conseguir con un ray tracing al estilo de Whitted (RTW).

Heckbert y Hanrahan [HH84] diseñan un algoritmo basado en RTW, pero trazando una pirámide (*beam*, según la terminología del artículo) por el plano de la cámara en lugar de rayos individuales. El resultado del recorrido son las secciones de las superficies de los objetos que intersecan con ese beam. Esto posibilita la realización de antialiasing más fácilmente debido a que las superficies representan la intersección de los infinitos rayos interiores a la pirámide.

Los dos trabajos anteriores no son exactamente un agrupamiento a priori ya que en ningún momento se usa ningún rayo interior a un cono o a un beam. Sin embargo, son trabajos pioneros en manejar directamente un volumen durante el recorrido, técnica que será usada posteriormente como método para acelerar el RT.

Arvo y Kirk [AK87] presentan una nueva EA jerárquica para el ray tracing. En cada nodo de esta estructura se divide la caja de la escena más el espacio de las direcciones, es decir, el hipercubo $\mathbb{R}^3 \times \mathcal{S}^2$ de cinco dimensiones. Cada dirección está especificada por el vector que comienza en el origen de coordenadas y llega a un punto de una cara de un cubo unidad centrado en el origen. Por tanto, dos coordenadas $(U, V) \in [0, 1]^2$, que corresponden a las coordenadas del punto de la cara, junto con la dimensión mayor de la dirección del rayo, son suficientes para especificar una dirección en el hipercubo.

En cada nodo interno, cada dimensión del hipercubo se divide en dos partes iguales, de manera semejante a un octree, aunque dejando 16 hijos en vez de 8, como es habitual en esta estructura. Así, cada nodo representa un volumen formado por una caja en el espacio más un conjunto de direcciones, lo que forma un volumen parecido a una pirámide. El recorrido de un rayo por esta estructura consiste en encontrar la hoja tal que su origen y dirección pertenezcan al hipercubo asociado a dicha hoja. Aunque, si los rayos son primarios, entonces el recorrido se puede hacer más sencillo.

Parker et al. [PSL+98] presentan un RT específico para el renderizado de *isosuperficies*. Dado un array tridimensional con datos de densidad, una isosuperficie consiste en una superficie que tiene la misma densidad en todos sus puntos. Los autores usan el propio array de datos como EA, recorriéndola con el algoritmo de Amanatides y Woo [AW87]. Si durante el recorrido se detecta que en una celda puede existir una sección de la isosuperficie, se realiza la intersección del rayo con la superficie resultante de interpolar los datos de densidad de los ocho vértices de la propia celda. En este trabajo ya se usa el agrupamiento a priori en función de la geometría de los rayos

primarios, junto con un reparto equilibrado de la carga.

El sistema anterior fue posteriormente mejorado por Parker et al. [PMS+99] para implementar un RTW. Este sistema es una de las primeras implementaciones de ray tracers que proporciona un rendimiento interactivo. A pesar de que el rendimiento es modesto, de unos pocos *frames per second* (*FPS*), ya se ven algunos conceptos que serán clave en el desarrollo posterior de los algoritmos de ray tracing interactivos en CPU.

Como medio de paralelismo, usan un sistema multiprocesador de hasta 64 procesadores. El sistema está organizado en un esquema maestro/esclavo con reparto de carga dirigido por la demanda. Cada tarea consiste en una cuadrícula (*tile*, en inglés) de 32×4 píxeles sobre la imagen final. Los procesadores se encargan de renderizar el color final de los píxeles de las cuadrículas lanzando a través de la escena varios rayos primarios por píxel.

En el reparto de carga se usan dos técnicas. La primera consiste en que dicho reparto no sea estático, sino guiado por la demanda. Cada tarea (un tile) requiere un tiempo diferente para ser terminada, ya que los rayos atraviesan diferentes zonas de la escena. Con un reparto dinámico, cada procesador esclavo que haya terminado de procesar su tarea pide una nueva al procesador maestro. Esto permite disminuir el tiempo total de renderizado al no mantener procesadores ociosos.

La segunda técnica consiste en agrupar las tareas por píxeles cercanos. Los rayos trazados sobre estos píxeles son muy parecidos, por lo tanto, aumenta la probabilidad de que la parte de la escena que necesita un rayo durante su recorrido sea también la misma que recorrerán los rayos posteriores sobre la misma cuadrícula. De esa manera se consigue un buen aprovechamiento del sistema de memoria.

Otra forma que los autores usan para aumentar el rendimiento consiste en disminuir el número total de rayos trazados sin perder la calidad de la imagen renderizada. Así, para implementar sombras con penumbra (o *soft shadow*) no usan un ray tracing distribuido (RTD), que tiene el inconveniente de que requiere el trazado de muchos rayos para que la apariencia de la sombra sea suave. Los autores optan en cambio por usar el método alternativo de Parker et al. [PSS98], que genera penumbras con solo un rayo de sombra. Sin embargo, este método es lineal en el número de objetos, por lo que solo es adecuado si el número de objetos de la escena no es muy grande.

En la parte de las conclusiones, los autores plantean cuestiones todavía sin resolver. Una es el uso de estructuras de aceleración específicas para escenas dinámicas. En este contexto, el tiempo de construcción de la estructura se convierte en una parte destacada en el rendimiento de la aplicación que puede no amortizarse con el posterior renderizado, como sí sucede con las escenas estáticas. Otra pregunta que lanzan es qué API gráfica sería la más adecuada para un ray tracing interactivo.

Wald et al. [WBWS01] presentan un RTW interactivo implementado en un PC convencional y en un clúster de PCs. Usan código altamente optimizado para la intersección rayo-triángulo, para el recorrido, y en el diseño de la capa de memoria. Los autores deciden usar triángulos como las únicas primitivas geométricas. Esto les permite optimizar el código de intersección rayo-triángulo, evitando, además, la instrucción de salto condicional (debido a sentencias *if-else* o a funciones virtuales), que resultaría imprescindible si se tuvieran diferentes tipos de objetos. Como estructura de aceleración usan un KD-Tree, argumentando que su algoritmo de recorrido requiere menos operaciones que otras estructuras (como, por ejemplo, las BVHs) y que se adapta mejor a la distribución de los triángulos por la escena que otras estructuras no jerárquicas (como, por ejemplo, una rejilla uniforme).

La memoria es el cuello de botella de su aplicación, por tanto, se realiza una cuidadosa implementación de la capa de memoria, tanto la del KD-Tree como la de los triángulos. En concreto, los nodos del KD-Tree están alineados en memoria, ocupando 8 bytes cada uno, lo que es suficiente para que 4 nodos quepan en una línea de cache.

La parte más interesante del artículo es aquella en la que se explica el uso de *paquetes de rayos*. Habitualmente, cada paquete está formado por los rayos primarios que forman un tile de

2×2 píxeles (un rayo por píxel), aunque es posible usar paquetes más grandes. En cada paso del recorrido, el paquete solo visita un nodo del KD-Tree cada vez, siendo este nodo común a todos sus rayos. El uso de paquetes tiene dos ventajas. Primero, la intersección de cada rayo con el plano del nodo puede hacerse en paralelo utilizando las operaciones SIMD de los procesadores. Segundo, una única petición a memoria sirve a cuatro rayos, disminuyendo el tráfico con memoria. Sin embargo, es necesario añadir código para manejar el caso en que no todos los rayos del paquete quieran visitar el mismo siguiente nodo. Los autores manejan estas divergencias mediante el uso de una máscara, evitando así que ciertas operaciones tengan éxito para algunos rayos.

El recorrido de estos paquetes de rayos es más rápido que el recorrido basado en rayos individuales cuando los rayos del paquete son coherentes. Los rayos primarios son geométricamente parecidos cuando poseen todos el mismo origen (en el artículo se utiliza una cámara *pinhole*) y tienen direcciones muy parecidas (cuadrícula de 2×2 píxeles). Con ello, todos los rayos del paquete van a estar activos y va a aumentar así la eficiencia de los accesos a memoria y de las operaciones SIMD.

Como se comenta en el propio artículo, la agrupación en paquetes a priori por criterios geométricos es sencilla cuando se consideran rayos primarios sobre píxeles cercanos. Sin embargo, los rayos secundarios generados a partir de la extensión de los primarios de un mismo paquete son menos coherentes. Concretamente, los autores implementan un RTW con rayos de sombra y de reflexión. Los rayos de sombra pueden trazarse de manera similar a los primarios si consideramos como origen de los rayos la posición de la luz y como destino los puntos de intersección de los rayos primarios. Para los rayos de reflexión los autores mantienen la misma agrupación que tenían los rayos primarios que los generaron.

Aparte del paralelismo que se encuentra dentro de la CPU, en forma de operaciones SIMD, los autores también implementan el sistema sobre un clúster de PCs conectados mediante una red Ethernet. El esquema de reparto de carga es similar al de Parker el al. [PMS+99], de forma que las tareas se mantienen en el procesador maestro y cada procesador esclavo pide una tarea nueva cuando acaba.

Wald et al. [WKB+02] siguieron desarrollando el sistema anterior introduciendo iluminación global. Para ello, dividieron la ecuación de renderizado en tres partes, de forma que cada una se ocupa de un efecto visual diferente: iluminación emisora, difusión por superficies diffuse, y cáusticas. Cada uno de estos efectos se aproxima usando una técnica diferente, con el objetivo de favorecer la coherencia de los rayos trazados.

La luz emisora es parte de la escena y solo depende del punto de intersección, por lo que solo se calcula su aportación cuando un rayo interseca con una superficie de una luminaria. Para la difusión de luz entre superficies, se usa la técnica de *Instant Radiosity* (A. Keller [Kel97]), donde una serie de puntos de luz virtual (o *VLPs*) son colocados por la escena. Cada uno de estos puntos de luz virtuales actúa igual que una luz, por lo que es necesario lanzar rayos de sombra hacia ellos. Para las cáusticas se usa *Photon Mapping* (H. Jensen [Jen96]), donde la irradiancia en un punto se obtiene consultando la densidad de los fotones más cercanos a dicho punto. La inclusión de los sistemas de iluminación global mediante Instant Radiosity y Photon Mapping añade una carga extra al sistema. Sin embargo, el número de VLPs y de fotones trazados es pequeño con respecto al número total de rayos, por lo que el tiempo dedicado a construirlos se amortiza sobre el renderizado total.

En general, los VLPs y los fotones son parte de la escena y, por tanto, comunes a todos los ordenadores clientes. Para evitar que esos datos tengan que replicarse, cada cliente mantiene su propio conjunto diferente de fotones y VLPs. Esto permite que el número de VLPs y de fotones se mantenga bajo, pero tiene la contrapartida de que en las imágenes se percibe correlación entre píxeles. El uso de *Interleaved Sampling* (Keller y Heidrich [KH01]) junto con *discontinued buffer* sirven para suavizar ese efecto.

Dietrich et al. [DWBS03] desarrollaron una API, llamada OpenRT, orientada al desarrollo

de aplicaciones de ray tracing interactivas en CPU. En los sistemas de rasterizado, tales como OpenGL, cuando un triángulo (o lote de triángulos) está listo para ser renderizado se manda a la GPU. Después, se liberan todos los recursos asociados con el triángulo. Este sistema no es adecuado para RT, ya que cualquier parte de la escena puede ser consultada durante el renderizado, por lo que toda la información tiene que estar disponible todo el tiempo.

Reshetov et al. [RSH05] desarrollaron una nueva forma de usar los paquetes de rayos para recorrer un KD-Tree. La idea es usar un *frustum* (o pirámide truncada) que encierre todos los rayos del paquete para disminuir el número de intersecciones rayo-caja. De esta forma, si no existe intersección entre el frustum y una caja, tampoco la habrá entre los rayos interiores al frustum y esa caja.

El recorrido del KD-Tree se divide en dos fases. En la primera, el frustum de un conjunto de rayos primarios se usa para encontrar el nodo de comienzo (*entry point*) de la fase siguiente. Para aumentar la eficiencia, el frustum puede dividirse para obtener puntos de entrada más profundos. Ya que cada frustum está formado a partir de los rayos primarios y de sombra, la división de un frustum se consigue rápidamente dividiendo la cuadrícula de sus rayos. La segunda fase es un recorrido basado en paquetes, similar al de los trabajos anteriormente descritos, pero comenzando desde el entry point obtenido en la fase anterior.

A. Reshetov [Res06] desarrolla un algoritmo de recorrido de KD-Trees con paquetes en el que no existe ninguna restricción sobre las direcciones o los orígenes de los rayos que lo forman. En trabajos anteriores, el recorrido con paquetes por un KD-Tree requería que las componentes de las direcciones de cada rayo tuvieran el mismo signo. De esa forma, el orden de recorrido de los hijos de un nodo era el mismo para todos los rayos del paquete. Si la restricción sobre los signos no se impone, entonces puede existir un caso en el que dos rayos del paquete tengan que recorrer los hijos de un nodo en orden diferente. En este trabajo, se realiza una encuesta a los rayos del paquete de forma que aquel nodo que vaya a ser recorrido más veces primero será el siguiente nodo del paquete, mientras que el otro se apila.

Aunque este sistema permite más oportunidades a la hora de empaquetar rayos, el recorrido es más costoso ya que es necesario realizar más consultas sobre los rayos. Experimentalmente, en un RTW se demuestra que el número de veces que hay que realizar la consulta no es muy grande, lo que permite un aumento en el rendimiento.

Boulos et al. [BEL+07] analizan el efecto que sobre el rendimiento tiene la reagrupación de los rayos secundarios en paquetes. Los sistemas de RT usados son RTW y RTD y la estructura de aceleración es una BVH. La manera de agrupar los rayos secundarios es por tipo, es decir, que los rayos de sombra y los de reflexión se trazan en dos paquetes diferentes. Los resultados muestran que siempre se mejora el rendimiento agrupando los rayos secundarios que no haciéndolo, aunque estos sean menos coherentes que los primarios.

Månsson et al. [MMAM07] implementan varias heurísticas para reorganizar dinámicamente los rayos en paquetes. En su implementación se mantiene un conjunto de rayos en la caché de rayos. Algunos de estos son posteriormente extraídos, agrupados en paquetes en función de sus propiedades geométricas y, finalmente, lanzados al recorrido. Desgraciadamente, su implementación en CPU usando KD-Trees no obtiene ninguna mejora con respecto a un recorrido sin paquetes.

Wald et al. [WBS07] investigan el uso de paquetes de rayos para el recorrido de una BVH. El algoritmo de recorrido empleado es como sigue. Una vez que está formado un paquete, se prueba la intersección del primero de sus rayos con la caja de un nodo (*early hit test*). En caso de que exista intersección, se considera que existe intersección de todo el paquete con el nodo. Si no existe, entonces se prueba la intersección del frustum del paquete con el nodo (*early miss test*). Un fallo en este test indica con seguridad que ningún rayo del paquete interseca el nodo. Si los dos tests anteriores han fallado, entonces se prueba la intersección individual rayo-caja de todos los rayos del paquete (*packet test of last resort*). Tan pronto como se encuentre un rayo del paquete que interseque la caja, se considera que el paquete interseca el nodo y ese rayo se guarda

como nuevo primer rayo del paquete (*first active ray tracking*). Este algoritmo de recorrido permite empaquetar rayos sin ninguna restricción, a diferencia de otros algoritmos para KD-Trees (Wald et al. [WBWS01]) que exigían direcciones con el mismo signo.

En una BVH, el orden de recorrido de los hijos de un nodo se determina habitualmente realizando la intersección con sus cajas y ordenándolos en función de su t_{entry}. En este trabajo, ya que solo se realiza una intersección rayo-caja en cada paso de recorrido, el orden de recorrido de los nodos hijo se determina heurísticamente en función del eje en el que las cajas tienen menor solapamiento.

Aunque se usan operaciones SIMD sobre el paquete de rayos, el algoritmo presentado en este trabajo no añade técnicas nuevas para aprovechar el paralelismo del ray tracing, sino mecanismos para reducir el número de intersecciones rayo-caja, que también sirven para disminuir el tiempo de procesamiento secuencial. Los mejores resultados se obtienen para paquetes de 8×8 y de 16×16 rayos primarios, obteniéndose beneficios en rendimiento desde $3.3\times$ hasta $10.7\times$.

Overbeck et al. [ORM08] estudian el rendimiento de tres algoritmos de recorrido de BVHs en función del tamaño de los paquetes. *Masked traversal* es la versión para BVHs del primer algoritmo de recorrido con paquetes para KD-Trees (Wald et al. [WBWS01]). La idea consiste en realizar la intersección de cada rayo del paquete con las cajas de los nodos de la BVH. Cuando se encuentra la primera intersección, se considera que el paquete interseca con ese nodo, convirtiéndose en el siguiente nodo que visitará el paquete. En *ranged traversal* (Wald et al. [WBS07]) también se actualiza el primer rayo que evalúa el test de intersección con éxito, evitando realizar sucesivos tests innecesarios. Así, se mantiene un intervalo de rayos dentro del paquete de forma que los rayos fuera del intervalo no han intersecado con la caja, pero sí al menos uno de los interiores. En *partition traversal* (aportación novedosa de este trabajo), al igual que en *ranged traversal*, se mantiene un intervalo de rayos pero se asegura que todos los rayos del intervalo intersecan con la caja, realizando las intersecciones rayo-caja para todos ellos. En los tres algoritmos se realiza el test *frustum culling* antes de comprobar las intersecciones rayo-caja.

Masked traversal solo es adecuado para rayos primarios y paquetes pequeños. *Ranged traversal* es adecuado para rayos primarios y secundarios, pero su rendimiento cae cuando el tamaño del paquete crece. *Partition traversal* es el algoritmo que da peores resultados en general, aunque es el más robusto con respecto a los rayos secundarios y diversos tamaños de paquete. Los autores exponen que el frustum culling es muy beneficioso para rayos primarios, en concreto, obtienen una ganancia entre $3.3\times$ y $6\times$.

Moon et al. [MBK+10] proponen ordenar los rayos antes del recorrido en función de su punto de intersección más cercano. Los autores buscan mejorar la eficiencia de la caché de disco cuando la escena completa no cabe en memoria. Su sistema de RT usa una BVH como estructura de aceleración. Se propone guardar los puntos de intersección en una rejilla uniforme y ordenar los rayos según el código de Morton de las celdas de esa rejilla. Así, si un conjunto de rayos tiene puntos de intersección cercanos entonces es muy probable que los nodos recorridos sean también muy parecidos. De esa forma, el orden en que los rayos son enviados al recorrido mejora el uso de las cachés de disco.

Sin embargo, esta ordenación a priori no es posible ya que los puntos de intersección de los rayos solo están disponibles después del recorrido. Para ello, se realiza la intersección de los rayos con una versión simplificada de la escena. Esta intersección es más rápida y los puntos obtenidos son usados como aproximaciones de las intersecciones reales de los rayos. Los resultados experimentales muestran para un PT que los accesos a disco disminuyen en más de un 93 % para una memoria que solo puede mantener un 23.8 % de la escena, obteniéndose una ganancia de hasta $16\times$.

Boulos et al. [BWB08] cambian el algoritmo de recorrido de BVHs de Wald et al. [WBS07] por otro en el que no se realiza especulación sobre los rayos de un paquete. Usando *aritmética de intervalos* (Boulos et al. [BWS06]) se realizan los test *all hit* y *all miss* para determinar si, respectivamente, todos los rayos del paquete tienen o no intersección con una caja. En caso de que

los dos tests anteriores hayan fallado, se realiza la intersección de todos los rayos con la caja. Para evitar sobrecargas, si el número de rayos activos disminuye hasta ser menor del 50 %, entonces los rayos activos se copian a otro paquete y se continúa con el recorrido de ese nuevo paquete. Los resultados experimentales muestran que esta técnica es más beneficiosa, sobre todo, para rayos incoherentes generados con PT y superficies diffuse y glossy.

Agrupamiento por comportamiento

Pharr et al. [PKGH97] tienen como objetivo principal diseñar un sistema para renderizar escenas grandes que no caben en la memoria principal. El tiempo de ejecución de una implementación directa para una escena de estas características es demasiado lenta, ya que está dominada por los fallos de caché de disco. Cada uno de estos fallos implica traer datos desde el disco hasta la memoria principal, lo que resulta en una gran penalización. Los autores diseñan un algoritmo en el que los rayos se añaden a las colas asociadas a las celdas de una rejilla uniforme, esperando para intersecar con los objetos contenidos en ellas. Un planificador se encarga de activar las colas de las celdas, además de actualizar las cachés de disco con información geométrica para que se puedan realizar las intersecciones. Los resultados experimentales muestran que, guardando solo un 10 % de la escena en caché, se consiguen resultados casi tan buenos como si cupiera completamente en memoria. Este artículo es un ejemplo de agrupamiento por comportamiento para aprovechar la coherencia de los rayos, orientado a la disminución del tráfico entre disco y memoria.

Navrátil et al. [NFLM07] diseñan un algoritmo de recorrido de un KD-Tree para reducir el tráfico entre memoria principal y caché L2 de las CPUs, aprovechando la coherencia de comportamiento de los rayos. Primeramente se seleccionan los nodos de la estructura de aceleración tales cuyos subárboles quepan completamente en caché L2. Estos nodos reciben el nombre de *puntos de cola*. Durante el recorrido, si un rayo llega a un punto de cola, suspende su recorrido y se queda esperando en la cola de ese nodo. Cuando la cola está llena, los rayos recorren el subárbol correspondiente al nodo de la cola. El recorrido presentado en este artículo es un recorrido sin pila, por lo que, en caso de que no haya encontrado intersección en una hoja, el rayo vuelve a comenzar desde la raíz, al estilo del KD-Tree restart de Foley y Sugerman [FS05]. Los resultados para un RTW con reflexión diffuse muestran una disminución del tráfico de hasta 7.8×.

Otros

Wald et al. [WBB08] presentan un algoritmo de recorrido para rayos incoherentes aprovechando las operaciones SIMD del procesador, pero sin usar paquetes de rayos. En este caso, las operaciones SIMD se usan para realizar la intersección en paralelo de un único rayo con varias cajas o con varios triángulos. Ya que la implementación está destinada al procesador Larrabee (Seiler et al. [SCS+08]), cuyas operaciones vectoriales tienen una anchura de 16, este es el máximo número de intersecciones rayo-caja o rayo-triángulo que se pueden realizar en paralelo. Debido al modo en que se recorren las BVHs, para poder realizar 16 intersecciones rayo-caja en paralelo resulta fundamental que los nodos internos de la BVH tengan un número elevado de hijos, hasta un máximo de 16. Para ello, una vez construida la BVH binaria (sección 3.4) se colapsan varios de sus niveles.

De los 16 nodos para los que se ha probado la intersección en cada paso de recorrido, aquellos que no han sido intersecados son excluidos del recorrido mediante una compactación. El resto se concatena, junto con su t_{entry}, en la lista de nodos intersecados por el rayo. El siguiente nodo que será visitado es aquel con un t_{entry} menor, que se obtiene realizando una reducción sobre la lista anterior.

Los resultados simulados para este algoritmo se compararon con otros algoritmos de recorrido con paquetes. Para rayos secundarios incoherentes, como los que aparecen en path tracing, se recorren menos nodos debido a que este algoritmo no es especulativo, como sí lo son los basados

en paquetes. Sin embargo, estas intersecciones son más costosas ya que requieren mayor tráfico con memoria (en cada paso de recorrido hay que traerse hasta 16 cajas), y generan más fallos de caché. El número de intersecciones rayo-triángulo sí es mayor como consecuencia de que las hojas de la BVH son más grandes. Con respecto al rendimiento, se aprecia mayor velocidad comparada con los algoritmos de paquete.

Dammertz et al. [DHK08] presentan una implementación de un recorrido de un único rayo por una BVH cuyo factor de ramificación es mayor que 2. Esta estructura, llamada *QBVH* o *quad-BVH*, se obtiene eliminado los niveles pares de una BVH. Así, cada nodo contiene las cuatro cajas de sus cuatro hijos (como máximo), pudiéndose realizar la intersección de un rayo con cuatro cajas mediante las operaciones SIMD de la CPU. Además, esta eliminación de nodos permite un gran ahorro en memoria que, junto con una cuidada implementación de la capa de memoria, permite un aumento en el rendimiento.

Ernst y Greiner [EG08] presentan un trabajo muy parecido al de Dammertz et al. [DHK08]. En este trabajo, también se construyen BVHs cuyos nodos internos tienen cuatro hijos, mediante la construcción de una BVH binaria y el posterior plegado de algunos niveles. Sin embargo, para que todas las hojas (excepto quizás una) contengan cuatro triángulos, durante la construcción solo se permiten divisiones que dejen un número de triángulos que sea múltiplo de cuatro.

A. Tsakok [Tsa09] propone un algoritmo de recorrido sin paquetes sobre una BVH con un factor de ramificación de 4, al estilo de Ernst y Greiner [EG08]. En cada paso del recorrido, si el nodo es interno, cada rayo del lote realiza la intersección con las cajas de los cuatro hijos del nodo mediante una operación SIMD. El resultado de estas intersecciones son cuatro pilas de rayos. Así, un rayo se encuentra en la pila i-ésima si la intersección de ese rayo con el hijo i-ésimo del nodo ha tenido éxito. Si el nodo es hoja, entonces el lote se divide en grupos de cuatro rayos, que realizan la intersección con un único triángulo usando operaciones SIMD.

Hunt y Mark [HM08b] construyen una rejilla uniforme en el espacio de perspectiva. Primero, los objetos se transforman a ese espacio y posteriormente se construye una rejilla uniforme sobre ellos. Esa estructura permite algunas optimizaciones en el recorrido de los rayos que tengan un origen común. Así, usando como centro de proyección el origen común de los rayos, no va a existir intersección entre los rayos y los planos de división de los ejes X e Y, lo que supone simplificar el algoritmo. Además, los autores no realizan divisiones en el eje Z, lo que reduce el recorrido a la intersección de un rayo con la lista de triángulos de la celda a la que pertenece.

B. Mora [Mor11] propone un recorrido de un lote de rayos que no usa una EA explícitamente guardada. Aunque es cierto que no se usa una EA previamente generada, el algoritmo es una mezcla entre construcción y recorrido de KD-Trees, pero sin guardar explícitamente la estructura. Cada llamada recursiva recibe una caja más un conjunto de triángulos y rayos que la intersecan. Para obtener el plano divisor, no se realiza ninguna estimación basada en SAH ya que, como argumenta el autor, es demasiado costoso para un ray tracing interactivo. En su lugar, si el número de triángulos es menor que 10000, entonces el plano divisor se coloca en la mediana espacial sobre el eje más largo de la caja. En caso contrario, el plano se coloca en la media de los centroides de los triángulos. Una vez obtenido el plano divisor, se prueba la intersección de todos los rayos y los triángulos con las cajas de sus hijos y se realizan las correspondientes llamadas recursivas.

Este recorrido sin EA explícita tiene la ventaja de que el manejo de escenas dinámicas es directo. Además, las partes de la escena que no se visitan durante el recorrido (es decir, no son visibles por ningún rayo) no se construyen. Sin embargo, ya se han presentado otros trabajos sobre construcciones rápidas de EAs, por lo que sería necesario una comparación experimental para demostrar el interés de un recorrido sin EA.

4.4.2. Trabajos Relacionados en GPU

Los primeros sistemas de ray tracing implementados en GPU se realizaron con la técnica clásica de programación de las GPUs para un propósito general (GPGPU). Debido a la dificultad de programación de estos procesadores y a los escasos detalles de sus arquitecturas, el objetivo de los programadores fue trasladar directamente el algoritmo de ray tracing desde CPU a GPU para aprovechar el paralelismo de esta arquitectura. Así, Carr et al. [CHH02] presentan una de las primeras implementaciones que usan la GPU para acelerar el RT. El recorrido de cada rayo se realiza en CPU usando un octree como EA, mientras que los tests de intersección rayo-triángulo se realizan en GPU. De manera similar a Pharr et al. [PKGH97], los rayos se quedan en las celdas del octree en espera de realizar la intersección con los triángulos contenidos. Cuando un lote de rayos (concretamente 256 rayos) está completo, se envía a la GPU junto con la geometría de la celda. El paralelismo de la GPU se usa para acelerar la intersección del lote de rayos con un triángulo. Para ello, se dibuja un cuadrado en la pantalla y se ejecuta un píxel shader, en donde cada píxel realiza la intersección rayo-triángulo consultando la información del rayo desde una textura, y las coordenadas del triángulo, que se pasan como atributos de los vértices del cuadrado. El procesamiento de los triángulos es secuencial, siguiendo un enfoque multipasada.

La implementación de este sistema está limitada por dos factores. El primero es la dificultad de la programación de las GPUs de la época, que se tuvo que realizar mediante GPGPU clásica. El segundo es la lenta comunicación entre la CPU y la GPU, por lo que el envío de los datos a la GPU y la posterior recuperación de los resultados de las intersecciones por la CPU suponen una penalización significativa.

Los trabajos de T. Purcell [Pur04] fueron los primeros en realizar una implementación completa de un ray tracing en GPU, también mediante GPGPU clásica. Purcell et al. [PBMH02] desarrollan varios ray tracers sobre un procesador de flujos, cuya arquitectura es similar a la de las GPUs (en el artículo simulan una GeForce 3 a la que se han añadido algunas facilidades de programación). Los autores tuvieron que enfrentarse con muchos problemas de programación. Entre ellos, la ausencia de instrucciones condicionales (bucles y saltos), de operaciones sobre enteros y de escrituras en posiciones aleatorias de memoria. Estas dificultades impusieron dos decisiones de diseño sobre el algoritmo. La primera fue usar una rejilla uniforme como estructura de aceleración, por la sencillez de su recorrido. Debido a la imposibilidad de realizar saltos condicionales, la segunda decisión consistió en dividir la etapa de recorrido en dos partes: el recorrido de las celdas de la rejilla y la intersección con los triángulos. La ejecución de estas etapas se intercalaba hasta que todos los rayos habían encontrado su punto de intersección más cercano. La decisión de qué etapa ejecutar se tomaba en la CPU teniendo el cuenta los estados de todos los rayos.

Purcell et al. [PDC+05] extienden el sistema anterior para introducir iluminación global mediante photon mapping. Su sistema está completamente implementado en una GeForce FX 5900 Ultra. Al igual que en el trabajo anterior, la estructura usada es una rejilla uniforme, pero esta vez para realizar consultas sobre los fotones. Los autores proponen dos métodos para construir la rejilla en GPU. El primero consiste en trazar los fotones y ordenarlos posteriormente en función de la celda de la rejilla en la que se encuentren. Debido a la imposibilidad de realizar escrituras aleatorias a memoria en la etapa de raster, esta ordenación se lleva a cabo en GPU mediante una ordenación bitónica (K. Batcher [Bat68]). La segunda manera consiste en cambiar la posición de un punto en un vertex shader para imitar escrituras aleatorias. Esta técnica tiene el inconveniente de que la memoria requerida para los fotones debe estar preasignada, lo que impone un límite máximo al número de fotones por celda.

Foley y Sugerman [FS05] presentan dos algoritmos de recorrido sin pila en un KD-Tree: *KD-Tree restart* y *KD-Tree backtrack*. Debido a las restricciones de las GPUs del momento, la implementación se realizó de nuevo con GPGPU clásica siguiendo el esquema multipasada. Durante el recorrido de un KD-Tree, cada rayo mantiene el intervalo de intersección $[t_{ini}, t_{end}]$ con el nodo

actual. En *KD-Tree restart*, al ser un recorrido sin pila, el siguiente nodo de recorrido de cada rayo será siempre el hijo más cercano. Si un rayo llega a una hoja y no encuentra intersección, comenzará el recorrido de nuevo desde la raíz, pero actualizando su origen a t_{ini}. De esta manera, ese rayo no volverá a visitar esa hoja, sino que llegará a la siguiente en el sentido de su dirección. Este recorrido tiene el inconveniente de que se tienen que visitar muchos más nodos que con el recorrido clásico con pila.

En *KD-Tree backtrack*, al igual que en *KD-Tree restart*, el origen del rayo se desplaza a t_{ini} cuando no se encuentra intersección en una hoja. Sin embargo, el recorrido no comienza de nuevo desde la raíz, sino que el rayo asciende por el KD-Tree hasta encontrar el primer nodo cuya caja interseque con el nuevo intervalo del rayo. Para conseguir esto, es necesario añadir a cada nodo la caja que representa el nodo más un puntero hacia su nodo padre. Este algoritmo visita más nodos que el algoritmo de recorrido clásico aunque menos que *KD-Tree restart*. Sin embargo, es necesario añadir información por nodo, lo que aumenta su consumo de memoria.

Horn et al. [HSMH07] añaden tres mejoras al algoritmo *KD-tree restart* de Foley y Sugerman [FS05]. La primera consiste en el uso de paquetes de rayos durante el recorrido para aprovechar las operaciones SIMD de las GPUs de la época. La segunda, llamada *push-down*, consiste en cambiar el nodo, llamado *nodo restart*, en el que se comienza cuando en una hoja no se encuentra ninguna intersección, a diferencia de Foley y Sugerman [FS05], donde el nodo restart es siempre la raíz. En este artículo, el nodo restart se actualiza durante el recorrido, para apuntar al primer nodo cuyos dos hijos tienen que ser visitados. La tercera mejora es el uso de una *pila corta*, que mantiene solo algunos de los últimos nodos pendientes por procesar. En caso de inserción, si la pila corta se encuentra llena, los nodos de la base se descartan. Esto implica que no se puede conocer el siguiente nodo del recorrido cuando la pila corta esté vacía y se necesite desapilar, por lo que es necesario realizar un restart. La idea detrás del uso de esta pila corta es mantener solo los elementos próximos a la cima de la que sería la pila de recorrido clásica. De estas tres mejoras, la que más aumentó el rendimiento fue la incorporación de la pila corta, que permite recorrer solo un 3 % más de nodos con pocos requisitos menores de memoria.

Roger et al. [RAH07] proponen una manera alternativa de calcular la intersección más cercana de cada rayo: los objetos de la escena se encargan de recorren la EA de los rayos para encontrar aquellos que los intersecan. Así, en vez de organizar los objetos de la escena en una EA, los rayos se organizan en una estructura jerárquica, llamada *ray-space hierarchy*, similar a una BVH. La implementación se realizó en GPU usando GPGPU clásica, por lo que todos los datos involucrados en el proceso tuvieron que codificarse en texturas bidimensionales. La construcción de la EA se realiza de manera bottom-up, comenzando por las hojas, donde se guardan los rayos. Cada nodo interno posee un volumen recubridor, al estilo de las BVHs, que encierra el volumen de sus cuatro hijos, mientras que cada hoja contiene a cuatro rayos. El volumen recubridor elegido es un cono, ya que se adapta mejor a la forma de los grupos de rayos. No se ha desarrollado ninguna heurística para la construcción de estas jerarquías, por lo que su construcción consiste en agrupar los rayos de cuatro en cuatro, según su disposición en la textura bidimensional.

Para aprovechar el paralelismo, todos los objetos recorren la EA a la vez. El recorrido se realiza en anchura, por lo que se mantiene la lista de las parejas triángulo-nodo que intersecan. En cada paso de recorrido, se interseca cada triángulo de cada pareja con los cuatro hijos del nodo. El resultado es posteriormente compactado para eliminar aquellas parejas que no han tenido intersección. Al final del proceso se obtiene la lista de todos los triángulos junto con los rayos que los intersecan.

Aparte de las mejoras algorítmicas, el aumento de las facilidades de programación de las tarjetas de la época, como la incorporación de instrucciones de salto y operaciones sobre enteros, posibilitó una implementación más eficiente en GPU. Esto facilitó la realización del algoritmo en un solo kernel, implementado usando BrookGPU [BFH+04].

Con la llegada de CUDA, se pudieron realizar implementaciones más eficientes sobre GPU.

Así, Günther et al. [GPSS07] presentan un algoritmo basado en paquetes para recorrer una BVH con CUDA. En este trabajo, un paquete de rayos tiene el tamaño de un warp (32 rayos) para evitar el uso de costosas operaciones de sincronización. La pila de recorrido es común a todos los rayos del paquete y está implementada en memoria compartida. En cada paso del recorrido, el paquete completo solo recorre un nodo de la BVH. Si el nodo es hoja, se intersecan todos los rayos del paquete con los triángulos de esa hoja; mientras que si es interno, se intersecan con las cajas de sus dos hijos. Si solo un nodo hijo es intersecado por los rayos del paquete, entonces ese será el siguiente nodo del recorrido. Si los dos nodos hijo han sido intersecados (por los mismos o por diferentes rayos), el siguiente nodo para el paquete se resuelve realizando una reducción paralela a nivel de warp: el nodo que haya sido visitado más veces será el siguiente en ser recorrido, mientras que el otro se apila. Al igual que el recorrido clásico de una BVH, la exploración termina cuando la pila se queda vacía.

Popov et al. [PGSS07] presentan un algoritmo de recorrido sin pila sobre un KD-Tree usando paquetes de rayos. Todas las hojas del KD-Tree se aumentan con seis hilvanes, uno por cara de su caja asociada, para guiar el recorrido de los rayos. El siguiente nodo del paquete se elige en función de los siguientes nodos de los rayos. Si todos tienen el mismo siguiente nodo, entonces el paquete se irá por él; si no, el siguiente será el nodo a mayor profundidad en el árbol. Los autores compararon el rendimiento de un recorrido con paquetes y sin paquetes sobre una GeForce 8800 GTX. El uso de paquetes de rayos permite satisfacer el estricto patrón de accesos a memoria para conseguir lecturas y escrituras fusionadas, por lo que este algoritmo obtiene mayor rendimiento comparado con el recorrido individual por rayo, sobre esa GPU.

Parker et al. [PBD+10] presentan una API flexible, llamada OptiX, para programar algoritmos de RT, en la que el usuario puede incorporar sus propios programas de intersección o de shading. La escena está representada por un grafo dirigido cuyos nodos guardan información de tres tipos: geometría, material y transformación. Esta representación es suficientemente flexible para cualquier tipo de escena, tanto estáticas como dinámicas.

Tras la compilación de los programas del usuario, estos son enlazados en un único kernel con toda la funcionalidad del programa. El kernel está estructurado como una máquina de estados, siguiendo el formato de una instrucción `switch-case`. El reparto de carga es dinámico, al estilo de los hilos persistentes de Aila y Laine [AL09]. Para minimizar el número de divergencias dentro de un warp, los autores tratan de predecir heurísticamente el siguiente estado más frecuente de todos los hilos.

El precio a pagar por la flexibilidad de esta API es que el rendimiento cae entre un 25 % y un 35 % con respecto a un RT específicamente implementado. Ludvigsen y Elster [LE10] presentan información de uso de OptiX sobre tarjetas de cap. 1.3. Los resultados comparados con Aila y Laine [AL09] indican que las implementaciones resultantes son entre 3 y 5 veces más lentas.

Garanzha y Loop [GL10] presentan un algoritmo de recorrido primero-en-anchura para un frustum de rayos. El algoritmo está dividido en dos partes. En la primera, los rayos se agrupan a priori por criterios geométricos y se construyen frustums de grupos coherentes. En la segunda, los frustums obtenidos en la etapa anterior se usan para recorrer una BVH en anchura.

Primero, cada rayo calcula su valor *hash*. Para ello, la escena se divide en una rejilla uniforme y cada rayo calcula su valor hash en función de la celda en que se encuentra su origen. De esta forma, rayos con el mismo valor hash tendrán orígenes cercanos. Posteriormente, los rayos son ordenados con CUDA, obteniendo un array donde todos los rayos con un mismo valor hash aparecen contiguos. La formación de los paquetes se realiza entonces tomando conjuntos de rayos con el mismo valor hash, que se encuentran consecutivos en el array tras la ordenación. De cada paquete de rayos se calcula su frustum, que contiene todos los rayos del paquete.

La BVH que se ha usado en este trabajo es un árbol donde cada nodo tiene un máximo de ocho hijos. Esta BVH se obtiene colapsando dos de cada tres niveles de una BVH binaria construida con el algoritmo top-down y SAH. El recorrido de los frustums por la BVH se realiza primero-en-

anchura. Cada frustum mantiene una cola de nodos con los que interseca. En cada nivel, se realiza la intersección de cada frustum con los hijos de cada nodo de su cola. Aquellos con los que interseca se meten en la cola, determinando heurísticamente cual es su orden de más cercano a más lejano. Una vez que se ha alcanzado el nivel de las hojas, se realiza la intersección de los rayos de cada frustum con los triángulos de las hojas, usando la estrategia de los hilos persistentes para equilibrar la carga de trabajo. Debido a que las hojas se visitan aproximadamente en orden de profundidad espacial, es posible descartar algunas en esta etapa.

Sobre el algoritmo descrito se han realizado algunas mejoras para aumentar su rendimiento. En primer lugar, la ordenación de los rayos por valor hash no se realiza directamente con un radixsort en CUDA, sino usando el esquema *CSD* (de *compression–sorting–decompression*). Siguiendo este esquema, lo primero que se realiza es una compresión de los datos resumiendo segmentos de datos contiguos en el array con el mismo valor hash en parejas de la forma *(hash, número de elementos)*. La ordenación posterior se lleva a cabo sobre este array de parejas ya que, al ser más pequeño que el array original, requiere menos tiempo. Por último, se descomprimen los datos resumidos para obtener el array original ordenado. Según los autores, el esquema CSD de ordenación es el doble de rápido que una ordenación directa por radixsort.

La segunda técnica consiste en usar el frustum durante el recorrido en vez de los rayos contenidos en él. Esto amortiza las operaciones si los paquetes son grandes (en el artículo manejan paquetes con entre 128 y 512 rayos). Por tanto, una única operación frustum-caja es suficiente para descartar una caja, en vez de realizar todas las intersecciones rayo-caja para los rayos del paquete.

La tercera consiste en usar una BVH con ocho hijos como EA, en vez de una BVH binaria. Una BVH de estas características es menos profunda, por lo que el número de pasadas durante el recorrido primero-en-anchura es menor, con el inconveniente de que cada pasada es más costosa. Los autores han comprobado experimentalmente que compensa el uso de una BVH con ocho hijos, en vez de con dos, en un recorrido primero-en-anchura.

Los resultados de rendimiento se han obtenido para un ray casting con sombras suaves, aunque en la presentación del artículo también se muestran resultados para un path tracing. El artículo explica que los resultados son mejores que los obtenidos con el algoritmo *speculative persistent while-while* de Aila y Laine [AL09], aunque la etapa dominante es ahora la de intersección rayo-triángulo.

S. Laine [Lai10] presenta un algoritmo para recorrer una BVH sin usar una pila. Este algoritmo es una adaptación del KD-Tree restart (Foley y Sugerman [FS05]) adaptado a BVHs. En una BVH no es suficiente con avanzar el origen del rayo y comenzar el recorrido desde la raíz para encontrar la siguiente hoja que visitar, ya que es posible que existan solapamientos entre nodos. Por tanto, para evitar que se vuelva a consultar una hoja ya visitada, cada rayo guarda un bit por nivel del árbol. Estos datos son suficientes para guiar el recorrido y evitar pasar dos veces por la misma hoja.

El autor también implementa un recorrido usando una pila corta, al estimo de Horn et al. [HSMH07], para evitar la penalización del restart. La realización experimental de este algoritmo demuestra que un recorrido sin pila puede llegar a visitar más del doble de nodos que un recorrido con pila. Sin embargo, una pila corta —de hasta tres nodos— disminuye los nodos visitados hasta un extra de solo 5 %. Además, la pila corta puede ser implementada en registros, que son memorias de acceso muy rápido.

Hapala et al. [HDW+11] presentan un algoritmo de recorrido de una BVH sin pila, donde cada nodo de la BVH es aumentado con un puntero a su padre. Un algoritmo de tres estados controla el recorrido, bajando o subiendo por la jerarquía, en función del caso. Como todos los algoritmos de recorrido sin pila, aquí también aumenta el número de nodos visitados durante el recorrido en comparación con el recorrido con pila. Ese exceso de nodos y, por tanto, el aumento de las consultas a memoria, es suficiente para que su rendimiento no sea mejor que el recorrido clásico con pila.

Pajot et al. [PBPP11] presentan una implementación de un bidirectional path tracing híbrido entre CPU y GPU. La GPU se usa para descargar a la CPU de ciertos cómputos, entre los que incluyen la visibilidad entre dos puntos, el cálculo de las probabilidades y las BRDFs. La CPU permite flexibilidad en el proceso, lo que es adecuado para renderizadores orientados a producción. La CPU se encarga de generar un lote de rutas desde la cámara y otro desde las luces. Esos datos son enviados a la GPU, en donde se calcula la visibilidad entre todas las posibles parejas de rutas anteriores. Esto permite aumentar la carga de la GPU sin aumentar el cómputo de la CPU ni la cantidad de datos enviados. Los cómputos en CPU y GPU se realizan ambos en paralelo, por lo que todo el proceso relacionado con la GPU, incluido el envío de datos, se oculta.

4.4.3. Hardware Específico

Aprovechando el conocimiento que se ha adquirido a partir de trabajos sobre implementaciones eficientes en CPU y GPU, otra línea de investigación ha sido el desarrollo de hardware específico que implemente los algoritmos de RT. Así, Schmitter et al. [SWS02] desarrollan la arquitectura *SaarCOR*, cuyo diseño estuvo muy influenciado por la implementación software más eficiente de la época, debida a Wald et al. [WBWS01]. Esta influencia se concreta en el uso de un KD-Tree como estructura de aceleración, y en el uso de instrucciones SIMD para las operaciones comunes de los paquetes de rayos. El hardware está dividido en tres unidades de funcionalidad fija, que implementan etapas diferentes del RT: generador de rayos, unidad de recorrido e interfaz de acceso a memoria. La jerarquía de memoria está dividida en una memoria *on-board*, más tres cachés on-chip, cada una para un tipo diferente de datos. Al igual que en las GPUs actuales, las latencias de la memoria on-board se pueden ocultar con multithreading. Una simulación de este hardware[7] muestra que se podría alcanzar ray tracing al estilo de Whitted en tiempo real.

Posteriormente, este primer hardware se amplió con nuevas características. Schmittler et al. [SLS03] añaden un sistema de memoria virtual, en el que la escena se guarda en la memoria del host si no cabe completamente en la memoria on-board del hardware. Así, las cachés on-chip actúan como cachés L1, mientras que la memoria on-board actúa como caché L2. Experimentalmente, los autores demostraron que una cantidad pequeña de memoria on-board (entre 8MB y 64MB) es suficiente para mantener los datos necesarios para el recorrido de un lote de rayos.

Schmittler et al. [SWW+04] adaptan la arquitectura SaarCOR a escenas dinámicas. Así, cada objeto mantiene un KD-Tree para su geometría en su sistema de coordenadas local y, sobre las cajas que cubren estos objetos, se construye un KD-Tree para la escena completa en coordenadas globales. Todos estos KD-Trees componen la EA de la escena: el KD-Tree global en la parte superior de la jerarquía y los locales de los objetos, en las hojas. Durante el recorrido, cuando un rayo llega a un KD-Tree local, tiene que transformarse al sistema de referencia del objeto. Ya que las operaciones de transformación son muy frecuentes, diseñaron una unidad específica que aplica una transformación afín al origen y a la dirección del rayo.

Woop et al. [WSS05] cambian la unidad de generación de rayos por una unidad completamente programable. Esto permite ejecutar programas personalizados para generar los nuevos rayos, algo que está inspirado en la manera en que las GPUs ejecutan los programas de shading para los píxeles. El diseño de esta nueva unidad es muy parecido a la forma en que se ejecutan los hilos dentro de un warp: los rayos se ejecutan de manera SIMD y las divergencias son manejadas automáticamente por el hardware. Sin embargo, cada rayo tiene a su vez la posibilidad de ejecutar operaciones SIMD de anchura 4. Por tanto, ya que los paquetes son de 4 rayos y cada rayo puede ejecutar 4 operaciones vectoriales, las operaciones SIMD de esta nueva unidad tienen la forma de una matriz 4×4.

[7] Según los autores, los requisitos hardware en términos de transistores son similares a los de las GPUs de la época como, por ejemplo, una GeForce 3.

Desafortunadamente, los experimentos de todos los trabajos relacionados con el hardware SaarCOR se han realizado con un RTW. En este ray tracing, los rayos involucrados (primarios, sombra, reflexión y refracción) son bastante coherentes, por lo que sus resultados no pueden extrapolarse directamente a otros algoritmos de iluminación global, tales como un PT.

La empresa Caustic [caua] desarrolló un prototipo para implementar el algoritmo de ray tracing, llamado CausticOne [cau09]. Desgraciadamente, no se da ningún detalle de su arquitectura [caub] y el proyecto parece abandonado a día de hoy. La empresa SiliconArts [sil] también desarrolló un hardware con funcionalidad fija, llamado RayCore [ray]. Al igual que Caustic, tampoco se dispone de ningún detalle de su arquitectura, aunque su funcionalidad parece fija y orientada a un RTW.

Aila y Karras [AK10] diseñaron una arquitectura con la finalidad de disminuir el tráfico de memoria que origina un conjunto de rayos incoherentes. El ray tracing usado en los experimentos es de iluminación global con rutas de longitud 2 sobre superficies difusas. Así, una gran parte de la escena puede ser consultada durante el proceso de renderizado, aumentando por ello la cantidad total de memoria transferida.

La arquitectura sobre la que se basan los autores es muy parecida a la de Fermi. El chip está compuesto por varios procesadores, y cada uno está formado por 32 cores que se ejecutan de manera SIMD. Cada procesador, además, mantiene 32 warps para mulithreading, cada uno de 32 hilos. El algoritmo de recorrido usado es *persistent while-while* [AL09], donde cada rayo está asignado a un hilo y cada hilo mantiene su propia pila de recorrido.

La estructura de aceleración que usan es una BVH organizada en *treelets*. Un treelet es un conjunto conexo de nodos de la BVH que se guardan consecutivos en memoria. En este artículo, el tamaño de los treelets está limitado para que quepan en una línea de caché. El recorrido de la BVH comienza en el treelet que contiene la raíz. Cuando un rayo llega a un nodo en la frontera del treelet, este se guarda en la cola del treelet del siguiente nodo que visitará. Posteriormente, un planificador se encarga de asignar cada cola de rayos a un procesador para continuar el recorrido de dichos rayos. El tráfico completo entre memorias del sistema se debe al intercambio de treelets, a la lectura de los rayos de cada cola y, sobre todo, a las pilas de recorrido. El uso de treelets junto con una caché que mantiene la cima de las pilas de recorrido (una pila corta al estilo de Horn et al. [HSMH07]) disminuyen las transferencias de memoria hasta en un 80 %.

Seiler et al. [SCS+08] proponen un procesador paralelo de propósito general, llamado *Larrabee*, sobre el que han implementado la tubería gráfica completamente en software —excepto la unidad de texturas, que estaría integrada como una unidad específica. El chip está formado por varios cores interconectados mediante una red, más una memoria caché L2. Cada core posee unidades funcionales para ejecutar instrucciones escalares y vectoriales (de anchura 16). Esta arquitectura permite una completa programación de todas las etapas de la tubería gráfica, incluida la parte de rasterizado. Aunque el objetivo en el diseño de este hardware no es el ray tracing, los autores aprovechan su flexibilidad para implementar un RTW sobre él.

Gribble y Ramani [GR08] discuten los detalles de un hardware basado en flujos de rayos, orientado a potenciar la eficiencia de la unidad SIMD. Una versión preliminar de este artículo se puede encontrar en Wald et al. [WGBK07]. Ramani et al. [RGD09] proporcionan algunos detalles extra sobre esta arquitectura. El recorrido comienza llenando un array con una cuadrícula de rayos primarios. Posteriormente, todos los rayos del array prueban su intersección con la raíz de la BVH. Los rayos que sí han intersecado se colocan al principio del array y se convierten en los nuevos rayos activos, mientras que los inactivos se colocan al final. Así se sigue hasta que no queden rayos activos, lo que significa que hay que desapilar, o hasta que la pila se vacíe.

Todas las operaciones de recorrido, además de las de intersección o de shading, se pueden realizar usando las operaciones SIMD, ya que operan sobre un conjunto de rayos activos. Debido a que las operaciones de filtrado se han encargado de eliminar todos los rayos que no están activos, la eficiencia de las operaciones SIMD será casi completa. Solo cuando el número de rayos activos

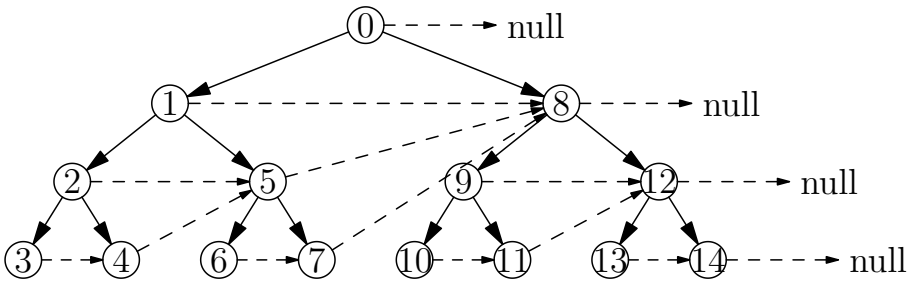


Figura 4.6: Ejemplo de una BVH hilvanada. Los hilvanes se muestran como líneas discontinuas.

caiga por debajo de la anchura de las operaciones SIMD, o cuando no sea múltiplo de esta, se perderá eficiencia. Además, también existe una pérdida en el rendimiento debido al sobrecoste de realizar continuamente la operación de filtrado.

Una simulación de este hardware muestra que, para rayos secundarios en PT y con anchura SIMD igual a 16, la eficiencia disminuye hasta un 38 % para el recorrido, y un 13 % para la intersección. El rendimiento de la simulación precisa de este hardware puede llegar hasta 26 FPS.

4.5. BVH Hilvanada

En esta sección presentamos nuestro trabajo [TMG09a], que consiste en un recorrido de una BVH sin el uso de pila. Para guiar el recorrido de los rayos, a cada nodo de la BVH se le ha añadido un puntero, llamado hilván. El hilván de un nodo n apunta al siguiente nodo que alcanzaría durante un recorrido del árbol en preorden y descartando de dicho recorrido el subárbol que tiene a n por raíz (figura 4.6). Además, grupos de rayos recorren juntos la BVH formando paquetes de rayos. Esto permite que se cumpla el estricto patrón de acceso a memoria que poseen las GPUs de cap. 1.x para que se produzcan lecturas fusionadas.

4.5.1. Recorrido de un Único Rayo por una BVH

El algoritmo de recorrido a través de una BVH hilvanada por un rayo se muestra en la figura 4.7. La variable `NR` (línea 3) indica el siguiente nodo de recorrido del rayo. Si un rayo no tiene intersección con la caja de un nodo interno, la variable `NR` se actualiza al nodo apuntado por el hilván del nodo actual (línea 12). En caso de que sí posea intersección, el siguiente nodo del rayo será su hijo izquierdo (línea 10).

Al igual que en el recorrido de una BVH con pila, si un nodo es hoja, se prueba la intersección del rayo con todos los triángulos contenidos en dicha hoja, actualizando consecuentemente el punto de intersección más cercano t_{min} (líneas 15–18). En este caso, el siguiente nodo de recorrido será siempre el apuntado por el hilván (línea 19). El algoritmo termina cuando se ha encontrado un hilván que apunta a `null` (línea 5), ya que no quedan nodos en la BVH por recorrer.

4.5.2. Recorrido de un Paquete de Rayos por una BVH

En las GPUs con cap. 1.0 y 1.1, el patrón de accesos a memoria necesario para que se produzcan lecturas fusionadas es muy estricto. Un recorrido con paquetes de rayos es el más adecuado para esta clase de GPUs ya que todos los rayos colaboran en la lectura de un nodo o de un triángulo, siguiendo el patrón correspondiente. El algoritmo de recorrido de una BVH hilvanada con paquetes de rayos se muestra en la figura 4.8. En este recorrido, al igual que en el recorrido por

```
1  // Inicialización de las variables.
2  float tmin = INFINITY;
3  Node  NR = root;
4  // Bucle de recorrido.
5  while( NR != null ) {
6    // Comprobamos si el nodo es hoja o interno.
7    if( NR es nodo interno ) {
8      intersección del rayo con NR;
9      if( hay intersección )
10       NR = NR.left;
11     else
12       NR = NR.rope;
13   } else {
14     // Se prueba la intersección de los triángulos de la hoja.
15     foreach( triángulo t en NR ) {
16       intersección del rayo con t;
17       actualizar tmin;
18     }
19     NR = NR.rope;
20   }
21 }
```

Figura 4.7: Recorrido de un rayo por una BVH hilvanada.

un único rayo, todos los rayos mantienen su siguiente nodo `NR` (línea 5). Además de `NR`, todos los rayos del paquete tienen la variable común `NP`[8], que indica el siguiente nodo del paquete (línea 6).

En la línea 11 todos los rayos del paquete colaboran para traerse el nodo apuntado por `NP`. Esto se consigue haciendo que cada rayo se traiga una sección de los bytes que ocupa el nodo `NP` en memoria, satisfaciendo el patrón de acceso a memoria para que este sea fusionado. Si el nodo es hoja, entonces los rayos del paquete también colaboran para traer los triángulos que contiene (línea 16).

Sin embargo, es posible que, para algún rayo, el siguiente nodo `NP` del paquete no coincida con el siguiente nodo `NR`. Decimos que un rayo está *activo* dentro de su paquete si su valor `NR` coincide con su valor `NP`. En otro caso, decimos que el rayo está *inactivo*. Los rayos activos son los únicos que realizan la intersección con su siguiente nodo de recorrido `NR`, ya que coincide con `NP`, que es el nodo traído de memoria. Por tanto, solo estos rayos pueden actualizar su `NR` y su t_{min}. Los rayos inactivos no actualizan su `NR` y se quedan "esperando" en su nodo hasta que vuelvan a estar activos. Sin embargo, tanto los rayos activos como los inactivos colaboran para realizar lecturas fusionadas en memoria.

El valor `NR` se actualiza de la misma forma que en el recorrido por un único rayo (líneas 20, 28 y 30). La actualización de `NP` se realiza a partir de los `NR` de los rayos activos del paquete. Si todos los rayos activos del paquete avanzan hacia el mismo nodo, entonces ese será el próximo nodo del paquete. Si algunos avanzan hacia el hijo izquierdo y otros por el hilván, entonces se presentan dos opciones para actualizar `NP`, aunque solo una es correcta. Si `NP` se actualiza al hilván, los rayos que se fueron por el hijo izquierdo se quedarán inactivos durante el resto del recorrido del paquete. Por el contrario, si `NP` se actualiza al hijo izquierdo, se tiene la garantía de que los rayos activos del paquete volverán a pasar por el nodo apuntado por el hilván, por lo que los rayos que se quedaron inactivos volverán a ser activos de nuevo.

La actualización de `NP` se realiza en las líneas 36–43. Primero, todos los rayos escriben en

[8] Esta variable en realidad está implementada en registros, por lo que cada rayo posee su propia copia de `NP`, aunque todos los rayos del paquete siempre actualizan su variable `NP` con el mismo valor. Por esta razón consideramos que `NP` es común a los rayos del paquete. Una alternativa hubiera sido implementarla en memoria compartida, en cuyo caso sería físicamente común.

```
1  // Memoria compartida.
2  __shared__ int left[PACKET_SIZE];
3  // Inicialización de las variables.
4  tmin = INFINITY;
5  Node NR = root;
6  Node NP = root;
7  // Bucle de recorrido.
8  while( NP != null ) {
9    // Comprobamos si el hilo está activo.
10   bool active = (NP == NR);
11   los rayos del paquete colaboran para traerse NP;
12   // Comprobamos si el nodo es hoja o interno
13   if( NP es un nodo hoja ) {
14     // Se prueba la intersección de los triángulos de la hoja .
15     foreach( triángulo t en NP ) {
16       los rayos del paquete colaboran para traerse t;
17       if( active ) {
18         intersecar el rayo con t;
19         actualizar tmin;
20         NR = NP.rope;
21       }
22     }
23   } else {
24     // Se prueba la intersección con la caja del nodo NP.
25     if( active ) {
26       probar intersección del rayo con NP;
27       if( hay intersección )
28         NR = NP.left;
29       else
30         NR = NP.rope;
31     }
32   }
33
34   // Cada rayo escribe 1 si se ha ido por el hijo izquierdo
35   // o 0 si se ha ido por el hilván.
36   left[rayID] = (NR == NP.left) ? 1 : 0;
37
38   // Realizamos una reducción sobre el array left con la operacion +.
39   // El resultado de la reducción se queda en left[0].
40   reduccion(left, +);
41
42   // Comprobamos el resultado de la reducción.
43   NP = (left[0] > 0) ? NP.left : NP.rope;
44 }
```

Figura 4.8: Recorrido de un paquete de rayos por una BVH hilvanada.

el array `left`, que se encuentra en memoria compartida (línea 36). Si el rayo se fue por el hijo izquierdo, entonces escribe 1, mientras que si se fue por el hilván escribe 0. Posteriormente, se realiza una reducción del array `left` con la operación + (línea 40). Si el resultado de la suma es mayor que 0, al menos un rayo del paquete ha avanzado por el hijo izquierdo, por lo que el siguiente nodo del paquete será el hijo izquierdo (línea 43). En caso de que la suma sea 0, `NP` se actualiza al hilván (línea 43). El recorrido termina cuando `NP` es `null` (línea 8).

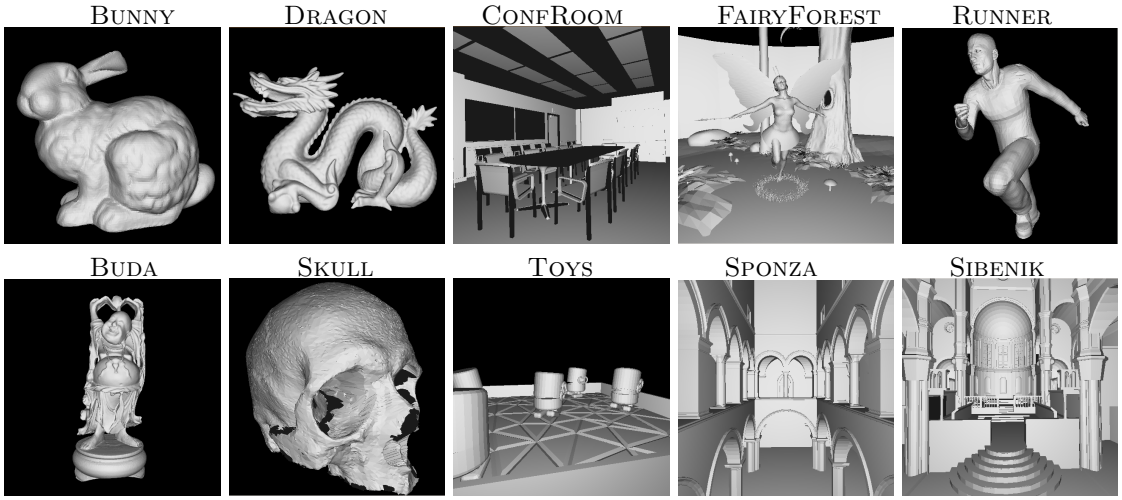


Figura 4.9: Capturas de las escenas usadas para probar el recorrido de una BVH hilvanada con paquetes.

4.5.3. Detalles de Implementación

Hemos implementado un sistema de ray tracing de rayos primarios para probar el rendimiento de una BVH hilvanada construida con SAH. Las GPUs que hemos usado son una NVidia GeForce 8800 GTS (cap. 1.0) y una NVidia GeForce 280 GTX (cap. 1.3). El sistema está implementado con tres kernels: *RayGenerator* (RG), *TraversalIntersection* (TI) y *Shading* (SH). El kernel RG se encarga de generar los rayos primarios (uno por píxel), mientras que TI encuentra el punto de intersección más cercano para estos rayos. El kernel TI es el más lento de los tres, consumiendo aproximadamente un 96 % del tiempo total. El test de intersección rayo-triángulo que se ha usado es el de Möller y Trumbore [MT97] y el test rayo-caja es el de Shirley y Morley [SM03]. El kernel SH calcula el color final del píxel usando el modelo de iluminación de Phong.

El algoritmo de recorrido que se ha implementado en TI es el recorrido de una BVH hilvanada con paquetes de rayos (sección 4.5.2). Este recorrido es el más adecuado para la GeForce 8800 GTS (cap. 1.0) debido al estricto patrón de acceso a memoria que requiere para realizar lecturas fusionadas. En este recorrido, cada rayo es manejado por un hilo y cada hilo está asociado únicamente a un rayo (no hay hilos persistentes). La memoria compartida se encarga de mantener la información común a los rayos del paquete, por tanto, solo hay dos alternativas para implementar un paquete: un paquete es un bloque de hilos o es un warp.

Si el paquete está implementado en un bloque de hilos usamos el nombre de `packet-block`. Por restricciones del hardware y de la implementación, los bloques pueden tener 256 hilos como máximo en la GeForce 8800 GTS, y 512 en la GeForce 280 GTX, lo que determina el número máximo de rayos por paquete. En concreto, el tamaño de los paquetes que hemos probado es de 16, 32, 64, 128, 256 y 512 rayos (este último solo para la GeForce 280 GTX).

Si el paquete está implementado en un warp usamos el nombre de `packet-warp`. Debido a que el tamaño del warp es siempre de 32 hilos, todos los paquetes tienen 32 rayos. En esta implementación, a diferencia de en `packet-block`, no es necesario el uso de barreras explícitas para la sincronización, debido a que los hilos de un warp se ejecutan simultáneamente.

La forma en que la BVH está guardada en memoria sigue el artículo de B. Smits [Smi98], es decir, la BVH está guardada en primero en profundidad. Por tanto, no es necesario que cada nodo interno mantenga un puntero al hijo izquierdo de cada nodo, ya que este siempre se encuentra en

Escenas	Triángulos	Geforce 280 GTX				Geforce 8800 GTS			
		`packet-warp`		`packet-block`		`packet-warp`		`packet-block`	
		FPS	forma	FPS	forma	FPS	forma	FPS	forma
Bunny	69,451	37.28	4×8	20.32	8×8	13.43	4×8	7.06	2×32
Dragon	871,414	21.60	8×4	10.35	8×8	6.73	4×8	3.17	4×16
Runner	78,029	41.67	2×16	19.80	1×64	16.91	4×8	9.34	2×32
FairyForest	174,117	22.55	4×8	11.82	2×32	7.97	4×8	4.25	2×32
Buda	1,087,716	24.33	2×16	9.51	1×64	8.65	4×8	3.92	2×32
ConfRoom	190,947	34.52	4×8	22.15	2×32	11.99	16×2	7.60	4×16
Sponza	66,454	26.49	8×4	15.97	8×8	8.55	4×8	4.95	2×32
Toys	11,141	50.72	4×8	33.23	8×8	19.80	4×8	12.14	4×16
Skull	102,905	29.64	4×8	14.32	2×32	10.32	4×8	5.15	2×32
Sibenik	80,479	23.74	4×8	14.79	8×8	7.56	4×8	4.51	4×16

Tabla 4.1: Mejores resultados de nuestro algoritmo de ray tracing para la GeForce 8800 GTS y la GeForce 280 GTX con una resolución de 1024×1024. En la columna "FPS" se muestran los *frames-per-second* de los mejores resultados en cada escena. En la columna "forma" se muestra la forma del paquete que proporcionó los resultados de la columna "FPS".

el nodo contiguo.

En nuestra implementación, cada lectura fusionada de memoria genera una transacción de 64 bytes. Cada triángulo ocupa 48 bytes, por lo que una transacción es suficiente para leer un triángulo completo. Un nodo de la BVH ocupa 32 bytes, por lo que una transacción es capaz de leer dos nodos. En nuestra implementación, cada vez que se lee un nodo también se trae de memoria el siguiente, que corresponde con su hijo izquierdo, si es un nodo interno. Si durante el recorrido, el siguiente nodo del paquete `NP` se actualiza al hijo izquierdo, entonces este ya se encuentra en memoria compartida, lo que ahorra una lectura de memoria.

4.5.4. Resultados

Hemos probado el algoritmo de ray tracing anteriormente descrito sobre las escenas de la figura 4.9. Las escenas Bunny, Dragon, Buda, Skull y Runner poseen un único objeto formado por gran cantidad de triángulos. Toys es una escena muy sencilla compuesta de pocos triángulos. Las escenas ConfRoom, FairyForest, Sponza y Sibenik representan entornos parcialmente cerrados. Todas las imágenes se han tomado con dos resoluciones: 512×512 y 1024×1024.

Los resultados de la aplicación para la resolución 1024×1024 se muestran en la tabla 4.1. Los resultados para la resolución 512×512 se pueden consultar en Torres et al. [TMG09a]. El rendimiento de la aplicación que se obtiene con `packet-warp` es siempre mejor que el que se obtiene con `packet-block`. Las razones por las que pensamos que esto sucede así son dos. Por un lado, porque `packet-warp` no usa sincronización explícita, por lo que no posee la sobrecarga debida a esta instrucción. Por otro lado, porque el tamaño de un paquete en *packet-warp* es siempre el número de hilos de un warp (32 rayos), se puede variar el número de warps que contienen los bloques de hilos y elegir el que ofrezca el mejor rendimiento experimentalmente.

Con respecto a `packet-block`, el tamaño de paquete que ofrece mejores resultados en la práctica es el de 64 rayos. Dos causas influyen en este resultado. Por un lado, si los bloques de hilos tienen un tamaño menor que 64 (es decir, 16 o 32 hilos) la ocupación del kernel TI disminuye significativamente por restricciones del hardware. Por otro lado, cuanto mayor es el bloque de hilos, mayor es también el paquete de rayos, lo que aumenta las probabilidades de que existan divergencias dentro del paquete y que el rendimiento caiga. Por tanto, bloques de 64 hilos suponen un buen compromiso entre el tamaño del paquete y la ocupación del kernel TI.

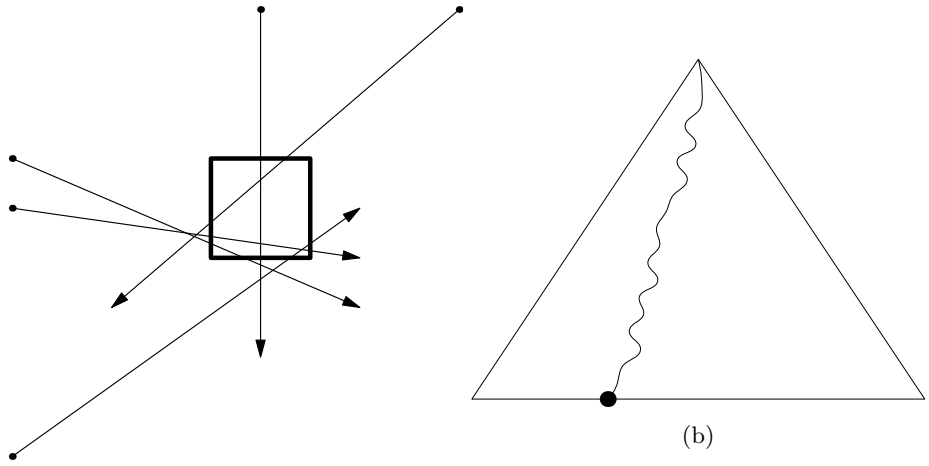


Figura 4.10: Fig. (a). Conjunto de varios rayos que intersecan una misma hoja de la BVH. Fig. (b). Los rayos de la figura (a) tienen en común el mismo camino desde la raíz de la BVH hasta la hoja.

4.6. Agrupamiento de una BVH Mediante un Corte

Como se ha visto en las secciones 4.2 y 4.4, lanzar al recorrido rayos que tengan orígenes y direcciones parecidos permite obtener un beneficio debido a la coherencia de esos rayos. Sin embargo, aunque los orígenes y direcciones de un conjunto de rayos sean muy diferentes, también pueden presentar coherencia si todos ellos intersecan la misma hoja (figura 4.10a). Esto se debe a que, al menos, existe un camino común de nodos para este conjunto de rayos durante el recorrido, que se corresponde con el camino que comienza en la raíz y termina en la hoja que todos intersecan (figura 4.10b).

Una forma de aprovechar esta coherencia sería eliminar todos aquellos rayos que con seguridad no intersecan esa hoja durante su recorrido. Esto se puede conseguir filtrando aquellos rayos que no intersecan algún nodo ancestro de dicha hoja. Para este fin, presentamos la noción de *corte* de una EA en nuestro artículo de Torres et al. [TMG11]. Un corte es un conjunto de subárboles de una EA que cumplen que cada hoja de la EA pertenece solamente a uno de estos subárboles (figura 4.11). Debido a cómo se realiza el recorrido de un lote de rayos por el corte (sección 4.6.1) usaremos una BVH como EA ya que cada nodo guarda explícitamente la caja que representa.

4.6.1. Recorrido de un Corte

El recorrido de un corte por un lote de rayos se realiza recorriendo por separado cada uno de los subárboles que lo componen. Primero, los rayos del lote prueban su intersección con la caja de la raíz de cada subárbol. Aquellos rayos que no intersecan con la caja de la raíz son descartados, mientras que el resto recorre cada subárbol usando cualquier algoritmo de recorrido en GPU. A la operación de descarte de rayos la llamaremos *filtrado*. En nuestro trabajo [TMG11] hemos probado dos algoritmos de recorrido dentro de cada subárbol: uno basado en paquetes (*persistent packet*) y otro de recorrido por rayo (*persistent while-while*), ambos debidos a Aila y Laine [AL09].

El recorrido entre los subárboles del corte se realiza secuencialmente y está controlado por la CPU. Esto tiene dos ventajas. Por un lado, no aumenta excesivamente la carga de memoria ya que el número de rayos filtrados es siempre menor o igual que el del lote inicial. Por otro lado, es posible realizar *early culling* para filtrar más rayos. Esto se debe a que, durante el recorrido de los

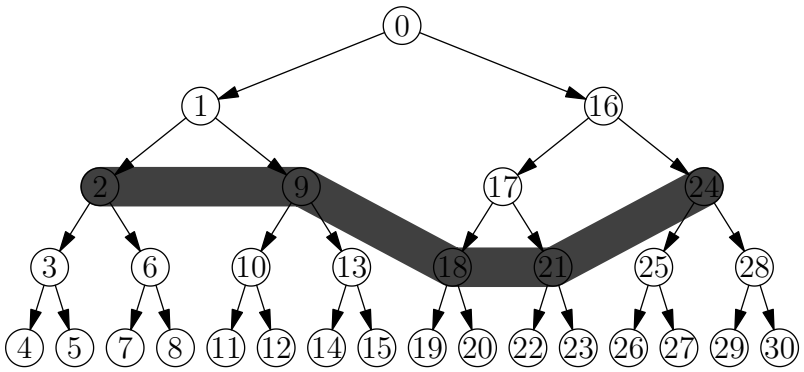


Figura 4.11: Ejemplo de un corte en una BVH. El corte está formado por los subárboles que cuelgan de los nodos 2, 9, 18, 21 y 24.

subárboles del corte, es posible que algunos rayos ya hayan encontrado al menos una intersección con algún triángulo, actualizando consecuentemente su t_{min}. Entonces, un rayo será filtrado del recorrido de un subárbol del corte si su valor t_{min} es menor que su punto t_{entry} de intersección con la caja de la raíz de dicho subárbol.

El beneficio en el recorrido de una BVH usando un corte es doble. Por un lado, el recorrido comienza siempre en nodos que se encuentran por debajo de la raíz, lo que evita el recorrido de los nodos que se encuentran entre la raíz y el corte. Por otro lado, el aprovechamiento de la memoria es mejor. Esto se debe a que la BVH está almacenada en primero-en-profundidad, es decir, los nodos de cada subárbol ocupan posiciones contiguas en memoria. Por tanto, recorrer un subárbol de un corte implica que disminuye el rango de direcciones de memoria que puede ser consultado, aumentando la probabilidad de que se produzcan lecturas fusionadas. Además, al ser el conjunto de posibles direcciones más pequeño, también aumenta la probabilidad de que se produzcan más aciertos de caché.

Los beneficios obtenidos por el uso de cortes aumentan a medida que las raíces de sus subárboles se encuentran a mayor profundidad en la BVH. Sin embargo, a mayor profundidad aumenta también la cantidad de subárboles que se tienen que recorrer, así como el sobrecoste, debido al filtrado, asociado a cada uno. Por tanto, es necesario comprobar experimentalmente si los beneficios del uso de un corte dado superan los inconvenientes que también conlleva.

4.6.2. Construcción de Cortes

Hemos usado dos técnicas de construcción de cortes: *construcción estructural* y *enfriamiento simulado*. En la construcción estructural, los nodos raíces de los subárboles se eligen teniendo en cuenta determinadas propiedades que solo pueden evaluarse en el contexto de la propia estructura. En el enfriamiento simulado, se realiza una búsqueda por el conjunto de cortes para obtener aquel con una estimación mínima del tiempo de recorrido.

Construcción estructural

En la construcción estructural, las raíces de los subárboles del corte se eligen teniendo en cuenta propiedades estructurales de la BVH. En este trabajo, hemos usado la profundidad de un nodo y el área de la superficie de su caja como parámetros en la construcción.

En la *construcción por profundidad*, el corte está formado por los subárboles cuyas raíces se encuentran a una cierta profundidad (figura 4.12a), siendo la profundidad elegida un parámetro

```
1  Cut create_depth(node n, int d) {
2    if(isLeaf(n) || depth(n) == d) {
3      return {n};
4    } else {
5      Cut C_l = create_depth(n.left, d);
6      Cut C_r = create_depth(n.right, d);
7      return set_union(C_l, C_r);
8    }
9  }
10
11 Cut create_cut(BVH bvh, int d) {
12   return create_depth(bvh.root, d);
13 }
```

(a)

```
1  Cut create_area(node n, float a) {
2    if(isLeaf(n) || SA(n) < a) {
3      return {n};
4    } else {
5      Cut C_l = create_area(n.left, a);
6      Cut C_r = create_area(n.right, a);
7      return set_union(C_l, C_r);
8    }
9  }
10
11 Cut create_cut(BVH bvh, float a) {
12   return create_area(bvh.root, a);
13 }
```

(b)

Figura 4.12: Algoritmos de construcción estructural de cortes por profundidad (a) y por área (b).

proporcionado por el usuario. Para evitar que se pierdan partes de la escena, si el algoritmo encuentra una hoja antes de llegar a la profundidad dada, esta se añade al corte, aunque no cumpla los criterios anteriores (línea 2). La *construcción por área* es muy parecida a la guiada por la profundidad (figura 4.12b). En este caso, las raíces de los subárboles del corte son los nodos menos profundos con un área —función `SA` en la línea 2— menor que un umbral de área proporcionado por el usuario.

Enfriamiento simulado

La cuestión que surge de manera natural es si existen cortes en una BVH que supongan una mejora en el rendimiento y que no sean los que construyen los algoritmos estructurales anteriores. Para responder esta pregunta trataremos de buscar otros cortes y encontraremos unos que, según veremos en la sección 4.6.4, tienen mejor rendimiento. Nuestro espacio de búsqueda consiste en el conjunto de todos los cortes posibles que se pueden formar sobre una BVH. Ya que este espacio es enorme, lo hemos simplificado de dos maneras. La primera consiste en buscar el corte solo entre los nodos cuya profundidad sea menor que un umbral dado. Esto descarta los cortes que se encuentran próximos a las hojas, cuyo tiempo de recorrido es alto. La segunda consiste en no tener en cuenta el orden de recorrido entre los subárboles del corte. Esto se consigue aproximando el tiempo total de recorrido de un corte como la suma de los tiempos de recorrido de cada uno de sus subárboles. Por tanto, esta aproximación es una cota superior del tiempo real de recorrido del corte ya que no se tiene en cuenta el early culling. Sin embargo, esto supone dos ventajas. Por un lado, el espacio de búsqueda es más pequeño ya que no se tienen que tener en cuenta todas las posibles secuencias de recorrido de los subárboles de cada corte. Por otro lado, se puede implementar una fase inicial en la que se guarda en cada nodo de la BVH el tiempo de recorrido junto con el filtrado de su subárbol, lo que libera al algoritmo de búsqueda de tener que calcularlos.

Aun con estas simplificaciones, el número de cortes de una BVH sigue siendo enorme, por lo que una búsqueda exhaustiva resulta irrealizable en la práctica. Por ello, hemos adaptado el algoritmo de búsqueda llamado *enfriamiento simulado* (Zomaya y Kazman [ZK99]) para encontrar un corte con un buen rendimiento. El espacio de búsqueda se expresa como un grafo de estados conexo y no dirigido, en el que cada estado tiene asociado un valor positivo, llamado *energía*. El enfriamiento simulado intenta encontrar el estado con mínima energía, moviéndose probabilísticamente entre estados adyacentes en función de la energía de cada estado y de la temperatura del sistema.

El algoritmo se comporta de la siguiente manera. De entre todos los estados adyacentes al actual, se elige uno al azar. Si su energía es menor, entonces se acepta sin condiciones y ese estado

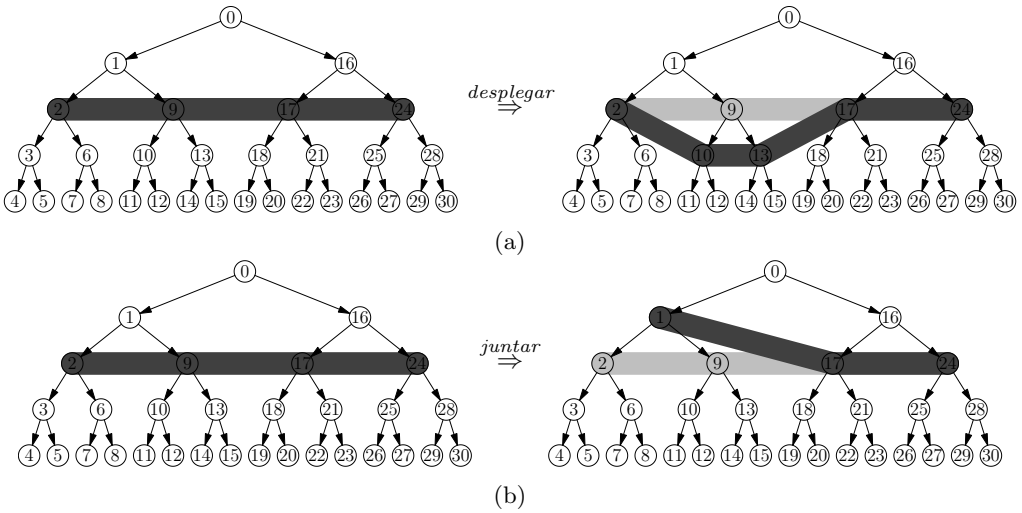


Figura 4.13: Fig. (a). Ejemplo de la operación *desplegar* en un corte: el nodo 9 ha sido sustituido por sus dos hijos, los nodos 10 y 13. Fig (b). Ejemplo de la operación *juntar* en un corte: los nodos 2 y 9 han sido sustituidos por su padre, el nodo 1.

adyacente se convierte en el actual. Si su energía es mayor, entonces se acepta con una cierta probabilidad p. Esta probabilidad debe cumplir que sea alta si la temperatura del sistema es alta o si la variación de la energía de los estados es pequeña, lo que evita que la búsqueda se quede atrapada en mínimos locales. La probabilidad que se suele usar es

$$p = min(1, e^{\frac{-|\Delta E|}{T}})$$

donde ΔE es la variación de energía entre los dos estados y T es la temperatura general del sistema.

Al principio del algoritmo, la temperatura del sistema es alta por lo que es muy probable que se acepten saltos entre estados con mucha diferencia de energía. A medida que la temperatura disminuye, es más difícil que se acepten transiciones a estados con más energía, lo que aumenta la probabilidad de que el estado actual se encuentre cerca del óptimo. Para aumentar la eficacia de la búsqueda, también se guarda el mínimo de todos los estados por los que se ha movido el enfriamiento simulado.

La adaptación de nuestro problema al esquema del enfriamiento simulado consiste en asignar un estado a cada corte. La energía de cada estado es la suma del tiempo que requiere un lote de rayos para filtrar y recorrer cada subárbol del corte. Dos estados son alcanzables entre sí si se puede obtener uno a partir del otro aplicando una de las dos operaciones siguientes: *desplegar* o *juntar* (figura 4.13). Desplegar consiste en sustituir un árbol de un corte por los subárboles de sus dos hijos. Juntar consiste en sustituir dos árboles cuyas raíces son nodos hermanos por el subárbol de su padre.

El esquema del enfriamiento simulado adaptado a la construcción de cortes se muestra en la figura 4.14. El estado de partida del enfriamiento simulado corresponde con el corte que contiene a toda la BVH (línea 3). El siguiente corte se obtiene aplicando la operación `evolve` al corte actual (líneas 7 y 24). La función `evolve` consiste en aplicar aleatoriamente una de las operaciones *desplegar* o *juntar* a un subárbol del corte, siempre y cuando sea posible. La temperatura del sistema comienza siendo `MAX_TEMP` (línea 9) y disminuye `NSteps` veces (línea 11). En cada iteración del bucle, la temperatura disminuye un porcentaje α de la temperatura anterior (línea 27). Para un cierto valor de temperatura, se intentan realizar `NSteps_per_temp` movimientos por el espacio de

```
1  Cut simulated_annealing(BVH bvh) {
2      // Estado inicial.
3      Cut currentCut = {bvh.root};
4      // Estado mejor.
5      Cut bestCut = currentCut;
6      // Estado siguiente.
7      Cut nextCut = evolve(currentCut);
8      // Temperatura inicial.
9      float temp = MAX_TEMP;
10     // Número de pasos de enfriamiento.
11     for(int i = 0; i < NSteps; i++) {
12         // Número de pasos en una misma temperatura.
13         for(int j = 0; j < NSteps_per_temp; j++) {
14             // Probabilidad de aceptación.
15             float p = exp(-|currentCut.time - nextCut.time| / temp);
16             // Realizar cambio de estado?
17             if( (nextCut.time < currentCut.time) || (rand(0,1) < p)) {
18                 currentCut = nextCut;
19                 // Actualizamos el mejor corte.
20                 if(currentCut.time < bestCut.time)
21                     bestCut = currentCut;
22             }
23             // Siguiente estado.
24             nextCut = evolve(currentCut);
25         }
26         // Disminuir la temperatura.
27         temp = α * temp;
28     }
29     return bestCut;
30 }
```

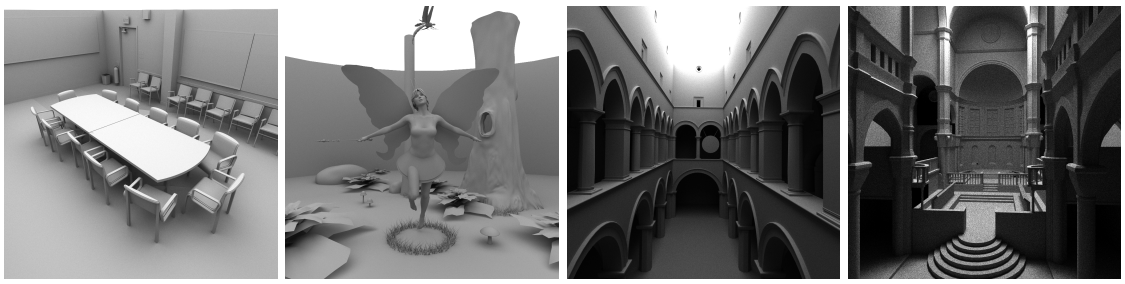
Figura 4.14: Construcción de un corte mediante enfriamiento simulado. La función *evolve* (línea 24) elige aleatoriamente entre las operaciones *desplegar* o *juntar*, siempre que se pueda.

estados (línea 13). Para cada posible movimiento, se calcula el valor de la probabilidad `p` (línea 15). Si el siguiente corte posee un tiempo de recorrido menor que el actual, entonces se acepta siempre. Si su tiempo de recorrido es mayor, el siguiente corte se acepta con probabilidad `p` (línea 17). Para mejorar la eficacia del algoritmo, se guarda el corte con menor tiempo de todos los que ha visitado el algoritmo de búsqueda (línea 20).

4.6.3. Detalles de Implementación

Hemos probado el rendimiento de una BVH recorrida mediante un corte sobre una NVIDIA GeForce GTX 285 (cap. 1.3) con 1GB de RAM. Las escenas que se han probado han sido FairyForest, ConfRoom, Sponza y Sibenik (figura 4.15). La escena FairyForest estaba abierta por arriba por lo que hemos colocado un techo para evitar que las rutas terminen al salir de la escena. Todas la imágenes se han tomado con una resolución de 1024×1024. La BVH usada se ha construido siguiendo el algoritmo top-down con SAH (sección 3.7), mejorando su rendimiento con la técnica de *early split clipping* de Ernst y Greiner [EG07]. Tanto los cortes como las BVHs usadas en nuestros experimentos se han construido en CPU.

Hemos implementado un Path Tracing (PT) sin ruleta rusa como algoritmo de renderizado y hemos establecido todos los materiales de las escenas a diffuse. Llamaremos *generación* al lote de rayos que han aplicado el mismo número de rebotes desde su origen en la cámara. Así, los rayos de la generación 0 forman el lote de los rayos primarios, los rayos de la generación 1, el lote de rayos



(a) (b) (c) (d)

Figura 4.15: Capturas de las escenas usadas en nuestros experimentos.

generados a partir de los primarios, y así sucesivamente. Las condiciones antes expuestas hacen que la coherencia del lote de rayos se degrade en cada rebote, llegando a exhibir un rendimiento muy bajo a partir de la generación 2. En la sección 4.6.4 analizaremos el uso de cortes para rayos desde la generación 0 hasta la 9.

La cantidad de rayos que se encuentra en un lote es el máximo que permite nuestra implementación y nuestra GPU: 8MRays ($= 8 \cdot 2^{20}$ rayos). Los rayos primarios se generan a partir de un array bidimensional de tamaño 4096×2048 sobre la película de una cámara. Ya que las imágenes tienen una resolución de 1024×1024, cada submatriz de 2×4 rayos contiene 8 muestras para un mismo píxel. Para almacenar el array bidimensional, se usa el código de Morton de las coordenadas de cada elemento. Por ello, cada submatriz de 2×4 ocupa posiciones contiguas en memoria.

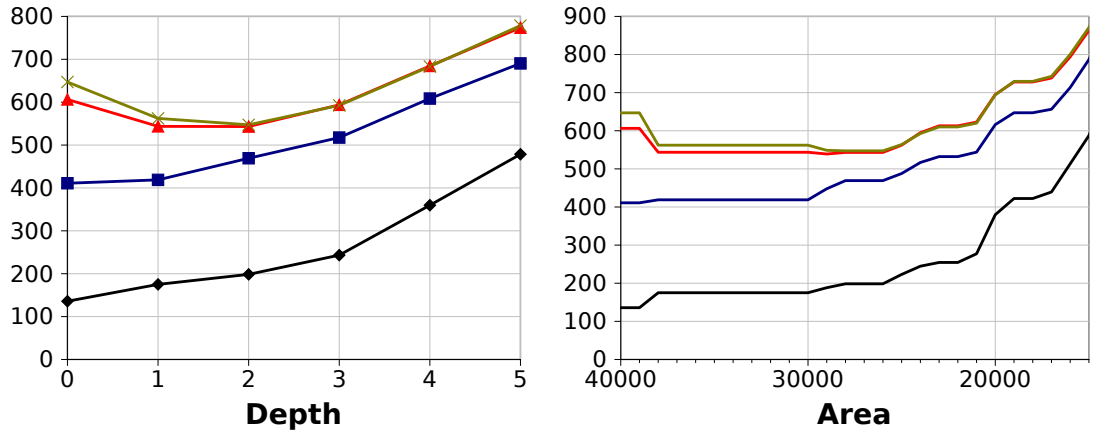
Para asegurar que no existe ninguna correlación entre rayos de la misma y de diferentes rutas, todos los números pseudoaleatorios que recibe cada rayo son diferentes. Hemos usado el generador de números pseudoaleatorios por congruencia lineal de Park y Miller [PM88]. El período de este generador es de $2^{31} - 2$, número que supera a la cantidad total de números aleatorios requeridos por todos los rayos.

El path tracing se ha implementado mediante cinco kernels de CUDA, ejecutados secuencialmente: *RayGenerator* (RG), *Test*, *Compact*, *Traversal-Intersection* (TI) y *Shading* (SH). Primero, los rayos primarios se generan desde la cámara en el kernel RG. Posteriormente, en Test, se prueba la intersección de los rayos con la caja de la raíz de un subárbol del corte. Los rayos que no tienen intersección con la caja se filtran en el kernel Compact. Este kernel está implementado con la primitiva *cudppCompact* de la librería CUDPP [HOS+08]. Posteriormente, el kernel TI encuentra el punto de intersección más cercano de cada rayo en el subárbol del corte. Los algoritmos usados como recorrido en GPU son *persistent packet* y *persistent while-while*, debidos a Aila y Laine [AL09]. El kernel SH se ejecuta tras haber realizado el recorrido de todos los subárboles del corte. Este kernel se encarga de generar el siguiente rayo de las rutas a partir de los puntos de intersección previamente encontrados.

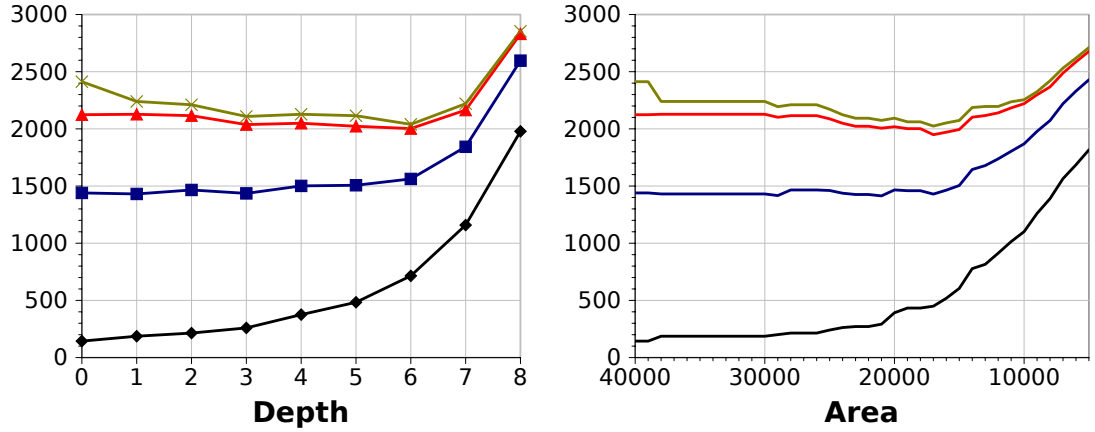
4.6.4. Resultados

Construcción estructural

Hemos probado el rendimiento de varios cortes estructurales construidos con diferentes umbrales de profundidad y de área. En esta sección solo mostraremos los resultados para la escena Sponza. Los resultados de las otras escenas se pueden consultar en Torres et al. [TMG11]. El tiempo de renderizado de los cortes para la escena Sponza se muestra en la figura 4.16. En el eje x se incluyen los parámetros de construcción de los cortes. En el eje y se muestra el tiempo completo de renderizado, medido en milisegundos (ms). Los puntos que corresponden a lotes de rayos de una



(a) Persistent while-while



(b) Persistent packet

Figura 4.16: Tiempos de renderizado (en *ms*) medidos para la escena SPONZA, usando cortes estructurales para recorrer una BVH. Los colores son: negro (generación 0), azul (generación 1), rojo (generación 2) y verde (generación 3).

misma generación se han unido por una línea continua. Aunque se han realizado experimentos para las primeras 10 generaciones, solo se muestran las generaciones del 0 al 3. El resto de generaciones tienen un comportamiento similar a la generación 3 y no se muestran por claridad.

El primer valor del parámetro (el situado más a la izquierda en las gráficas) corresponde a cortes con un único subárbol que cubre toda la escena, es decir, la propia BVH. A ese corte le hemos denominado Cut_{root}. Por tanto, el primer valor de cada curva indica el tiempo de renderizado con el recorrido de la BVH completa más el tiempo extra debido al filtrado. Ese tiempo extra está en torno a 10 ms según nuestros experimentos.

Las curvas de cada generación tienen formas similares en todas las escenas. Las generaciones 0 y 1 no experimentan ninguna mejora con respecto al recorrido de Cut_{root} (obsérvese que se trata de curvas crecientes). Esto se debe a que esos rayos son ya muy coherentes y la mejora obtenida lanzando paquetes algo más coherentes no es suficiente para amortizar la sobrecarga extra del filtrado. Por el contrario, las curvas de las generaciones 2 a 9 muestran un valle al comienzo y

		Generación									
Corte Estructural		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Corte en Profundidad	Ahorro (en %):	0.0	0.0	10.4	15.4	17.1	18.3	19.0	19.5	19.8	20.1
	Profundidad:	0	0	2	2	2	2	2	2	2	2
Corte en área	Ahorro (en %):	0.0	0.0	11.1	15.4	17.1	18.3	19.0	19.5	19.8	20.1
	Área (en %):	100	100	75.3	72.7	72.7	72.7	72.7	72.7	72.7	72.7

(a) **Persistent while-while**

		Generación									
Corte Estructural		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Corte en Profundidad	Ahorro (en %):	0.0	0.6	5.7	15.4	12.8	17.9	14.8	19.7	15.6	20.0
	Profundidad:	0	1	6	6	6	6	6	6	6	6
Corte en Área	Ahorro (en %):	0.0	1.7	8.2	16.0	13.6	19.7	15.4	20.0	16.8	19.7
	Área (en %):	100	54.5	44.1	44.1	44.1	44.1	44.1	44.1	44.1	44.1

(b) **Persistent packet**

Tabla 4.2: Resultados de los mejores cortes estructurales para la escena Sponza. Para cada generación se muestra el corte estructural por profundidad y área que tarda menos tiempo en renderizar la escena. Para cada uno se muestra su *Ahorro* (en %) con respecto a Cut_{root}, junto con la profundidad (si es un corte en profundidad) o el porcentaje de área con respecto a la raíz (si es un corte en área) que se han usado en su construcción.

sufren un crecimiento después. Ese crecimiento es exponencial y no se ha incluido completamente en las gráficas.

La ganancia de cada curva con respecto a Cut_{root} es más importante en persistent packet que en persistent while-while. Ya que, fijado un parámetro de construcción, la sobrecarga debida al filtrado es la misma para ambos recorridos, pensamos que el hardware debe de ser el responsable de esta diferencia. Si los grupos de rayos son más coherentes en persistent while-while, el número de nodos leídos de memoria no varía, pero existirán más oportunidades de realizar lecturas fusionadas. Si los paquetes en persistent packet son más coherentes, disminuirán las peticiones de nodos a memoria, siendo sus lecturas siempre fusionadas. El sistema de memoria de las GPUs permite que el beneficio debido a la disminución en el número de nodos leídos sea mayor que un aumento en las lecturas fusionadas.

Las tablas 4.2a y 4.2b resumen los mejores resultados para Sponza. Estas incluyen una columna por cada generación que muestran los porcentajes de ahorro de los mejores cortes estructurales con respecto a Cut_{root}[9]. Los porcentajes de ahorro más relevantes se encuentran en la escena Sibenik (30.6 % con profundidad y 32.0 % con área) para el recorrido persistent while-while, y en la escena FairyForest (22.7 % con profundidad y 40.9 % con área) para el recorrido persistent packet.

Enfriamiento simulado

Los resultados del enfriamiento simulado para Sponza pueden verse en la tabla 4.3. Los parámetros del algoritmo usados durante la búsqueda son `MAX_TEMP` = 600, `NSteps` = 1000, `NSteps_per_temp` = 1000 y $\alpha = 0.99$. En general, el algoritmo de enfriamiento simulado encuentra cortes con mayor rendimiento que no se corresponden con ninguno de los obtenidos con construcción estructural. Esto es lógico, ya que con este algoritmo se permite la construcción de un número

[9]Si, por ejemplo, se obtuviera un porcentaje de ahorro del 10 %, entonces el recorrido con corte tardaría 0.9× del recorrido de Cut_{root}.

	Generación									
Enfriamiento Simulado	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Ahorro (en %):	0.0	0.0	11.1	15.4	17.1	18.3	19.0	19.5	19.8	20.1
Profundidad media:	0.0	0.0	1.6	2.0	2.0	2.0	2.0	2.0	2.0	2.0
Área media (en %):	100	100	68.2	64.5	64.5	64.5	64.5	64.5	64.5	64.5

(a) **Persistent while-while**

	Generación									
Enfriamiento Simulado	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Ahorro (en %):	0.0	3.4	14.8	24.1	21.5	26.6	22.9	27.3	23.2	27.5
Profundidad media:	0.0	5.5	6.0	6.4	6.5	6.4	6.3	6.2	6.3	6.3
Área media (en %):	100	32.0	29.3	28.2	30.9	29.9	30.1	30.3	30.1	29.9

(b) **Persistent packet**

Tabla 4.3: Porcentaje de ahorro en el tiempo de renderizado del mejor corte encontrado por el algoritmo de enfriamiento simulado con respecto a Cut_{root} para la escena Sponza. También se muestran la profundidad y el área media (en % con respecto a la raíz de la BVH) de este mejor corte.

mayor de cortes que los que se generan con cualquiera de las construcciones estructurales.

El mejor resultado para persistent packet lo obtiene la escena FairyForest, con un ahorro del 51.7 %. La profundidad media del corte es de 5.9 y el área media es un 17.9 % del área de la raíz de la BVH. Para persistent while-while, la escena Sibenik es la que obtiene un mejor resultado con un ahorro del 32.0 % para un corte con profundidad media de 3.1 y porcentaje medio de área del 43.8 %.

4.6.5. Propuesta de Estimación del Coste de un Corte

De manera similar a como se hizo en la sección 3.5 para una estructura de aceleración jerárquica, podemos estimar el coste que tendrá un corte durante el renderizado de una escena. La intención al desarrollar una función de coste es, nuevamente, que sea usada en la construcción de cortes. El coste que proponemos a continuación es simplemente eso, una propuesta, y no ha sido implementado todavía por lo que tiene que evaluarse experimentalmente para demostrar su aplicación en la práctica.

Sea $Cut = \{n_1, \ldots, n_N\}$ un corte con N subárboles, donde cada nodo n_i es la raíz de un subárbol. Como se describió en la sección 4.6.1, el recorrido de una BVH con corte consiste en un recorrido secuencial de cada uno de sus subárboles. Cada uno de estos recorridos está divido en tres fases: una prueba de intersección de los rayos con la caja de su raíz, una compactación y, por último, un recorrido clásico de una BVH. Usaremos estas tres fases para determinar su coste. Así, definimos el coste de un corte como el producto de todos los rayos usados durante el renderizado de una escena por el coste medio del corte:

$$|ray_{root}| \cdot coste(Cut) = \sum_{i=1}^{N} \Big(|ray_{root}| \cdot C_{caja} + C_{comp}(n_i) + coste(n_i) \cdot |ray_{n_i}| \Big) \tag{4.5}$$

donde $|ray_{root}|$ es el número total de rayos usados durante el renderizado, $|ray_{n_i}|$ es el número de rayos que intersecan con el nodo n_i, $coste(Cut)$ es el coste medio por rayo del corte, $coste(n_i)$ es el coste medio por rayo del subárbol que tiene a n_i por raíz, C_{caja} es el coste de una intersección

rayo-caja, y $C_{comp}(n_i)$ es el coste de la compactación del nodo n_i.

El algoritmo de compactación que hemos usado (sección 2.4.2) está dividido en dos partes. En la primera se realiza un scan sobre todos los elementos, que es independiente del número de elementos que han pasado el test. La segunda sí depende exclusivamente de los elementos que han pasado el test y consiste en la escritura, en el array destino, de cada uno de ellos. Por ello, hemos dividido el coste de compactación C_{comp} en dos partes, una que depende de todos los rayos y otra que solo depende de los rayos que intersecan con la raíz:

$$C_{comp}(n_i) = |ray_{root}| \cdot C_{scan} + |ray_{n_i}| \cdot C_{addr}$$

donde C_{scan} es el coste medio de cada ejecución del scan, y C_{addr} es el coste de una escritura en memoria.

Siguiendo las mismas aproximaciones que sobre los rayos se hicieron para el coste SAH (sección 3.5.1) podemos aproximar la fracción $\frac{|ray_{n_i}|}{|ray_{root}|}$ como el área de la superficie de n_i dividida por el área de la superficie de la caja de la escena. Así, la estimación del coste medio de recorrido de un corte, que se obtiene despejando el coste medio en la ecuación 4.5, y aplicando las mismas suposiciones que SAH, queda como

$$\hat{coste}(Cut) = N(C_{caja} + C_{scan}) + \sum_{i=1}^{N} \frac{SA(n_i)}{SA(root)} C_{addr} + \sum_{i=1}^{N} \frac{SA(n_i)}{SA(root)} coste(n_i) \qquad (4.6)$$

Obsérvese que el árbol que tiene por raíz al nodo n_i ya está construido, por lo que es posible calcular su coste medio (ecuación 3.1) aproximando la probabilidad de intersección como la relación entre áreas de superficies. Obsérvese también que esta estimación de coste es una cota superior ya que no se ha tenido en cuenta el *early culling* de las BVHs.

Al igual que para el coste de una EA, los valores de las constantes determinan las unidades de medida de este coste. Si buscamos minimicar el número de operaciones, entonces establecemos las constantes $C_{caja}, C_{scan}, C_{addr}, C_t$ a 1 y C_i a 2. Las constantes C_t y C_i son el coste de intersección de un rayo con un triángulo y con un nodo interno respectivamente, como se explicó en la sección 3.5.

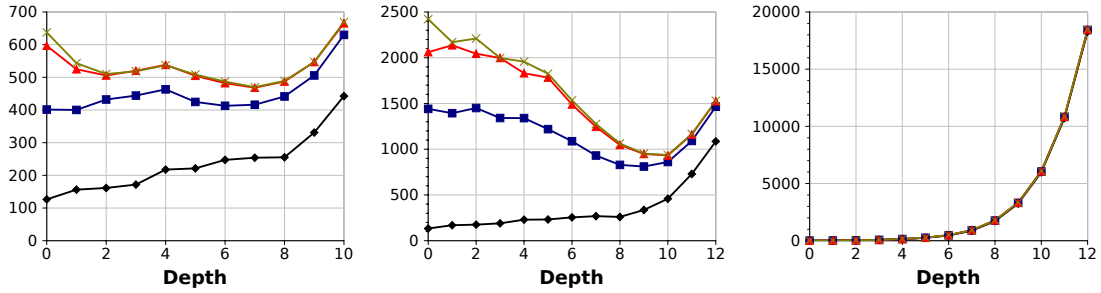
Veamos cómo cambia el coste de un corte *Cut* a medida que se le aplica la operación *desplegar* sobre alguno de sus nodos. Si observamos el primer sumando de la ecuación 4.6, vemos que siempre aumenta, ya que solo depende del número de subárboles del corte. El segundo sumando puede aumentar o disminuir, dependiendo de las áreas de las superficies de los nuevos nodos introducidos en el corte. El tercer sumando siempre decrece ya que existe ahorro al recorrer subárboles a mayor profundidad. Por tanto, es posible que un corte más profundo tenga un coste menor si el valor agregado de los sumandos que aumentan es menor que el valor de los sumandos que decrecen.

Podemos ver esto para el caso del corte $Cut' = desplegar(Cut, n_j)$, un corte al que se le ha aplicado la operación *desplegar* sobre el nodo n_j. Este corte es $Cut' = (Cut \setminus \{n_j\}) \cup \{l, r\}$, donde l y r son los hijos izquierdo y derecho, respectivamente, del nodo n_j. En general, el coste de Cut' es menor que el de Cut si se cumple que

$$\begin{aligned}
& \hat{coste}(Cut') < \hat{coste}(Cut) \\
\Leftrightarrow\ & C_{caja} + C_{scan} + P(l) \cdot C_{addr} + P(r) \cdot C_{addr} + P(l) \cdot coste(l) + P(r) \cdot coste(r) < \\
& \quad P(n_j) \cdot C_{addr} + P(n_j) \cdot coste(n_j) \\
\Leftrightarrow\ & C_{caja} + C_{scan} + P(l) \cdot C_{addr} + P(r) \cdot C_{addr} < P(n_j) \cdot C_{addr} + P(n_j) \cdot C_i
\end{aligned}$$

donde $P(n) = \frac{SA(n)}{SA(root)}$.

Para construir un buen corte a partir de una EA ya generada, sería interesante un algoritmo que encontrara el corte con coste mínimo. Al igual que en la construcción de EAs, se puede usar un algoritmo devorador que comience en Cut_{root} y que despliegue nodos en función de decisiones



(a) (b) (c)

Figura 4.17: Fig. (a). Tiempo de renderizado sin la sobrecarga del filtrado para la escena Sponza usando persistent while-while. Fig. (b). Tiempo de renderizado sin la sobrecarga del filtrado para la escena Sponza usando persistent packet. Fig. (c). Tiempo debido solo al filtrado.

locales. Por ejemplo, dado un corte, el algoritmo puede probar el coste de todos los cortes alcanzables desde el actual aplicando la operación *desplegar*, y quedarse con aquel con menor coste, prosiguiendo hasta que no exista ningún corte alcanzable con menor coste que el actual.

4.6.6. Propuesta de Reducción de la Sobrecarga del Uso de Cortes

El beneficio en el uso de cortes para recorrer BVHs es consecuencia de que la sobrecarga debida al filtrado (kernels *Test* y *Compact*) es menor que la mejora obtenida por el recorrido de rayos más coherentes. En las figuras 4.17a y 4.17b se muestran los tiempos de renderizado para la escena Sponza sin la sobrecarga del filtrado. Como se puede ver en estas gráficas, existe todavía suficiente coherencia dentro de un lote de rayos como para permitir la disminución del tiempo de renderizado completo. Sin embargo, esta coherencia no aumenta el rendimiento general debido a que la sobrecarga del filtrado crece exponencialmente con la profundidad del corte (figura 4.17c). Por tanto, el filtrado es un factor limitador muy importante y cualquier técnica orientada a la reducción de esta sobrecarga aumentaría más todavía el rendimiento de la aplicación.

Una de las técnicas con las que hemos tratado de disminuir la sobrecarga del filtrado es con el uso de dos cortes Cut_t y Cut_f. Estos dos cortes deben cumplir que para todo nodo raíz $m \in Cut_t$ exista solo un nodo raíz $n \in Cut_f$ de manera que n se encuentre en el camino desde m a la raíz de la BVH (figura 4.18). Esto implica que todo nodo raíz Cut_f tiene "por debajo" un conjunto de

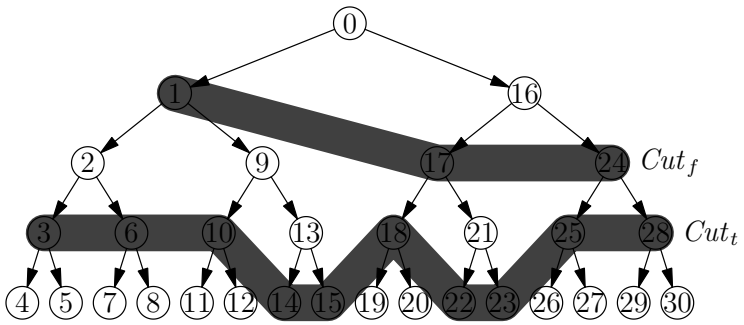


Figura 4.18: Dos cortes sobre una BVH. El corte Cut_f se usa para filtrar los rayos, mientras que el corte Cut_t se usa para recorrer la BVH.

nodos del corte Cut_t. Esto nos permite repartir las tareas del recorrido entre estos dos cortes. El corte Cut_f se usa solamente para filtrar el lote de rayos. Una vez filtrados para un cierto nodo de Cut_f, el conjunto de rayos resultante recorre los subárboles de Cut_t que están por debajo de ese nodo.

Un recorrido de una BVH usando dos cortes tiene la ventaja de que el número de operaciones de filtrado disminuye, ya que el corte Cut_f se encuentra por encima de Cut_t. Sin embargo, este filtrado es menos estricto y puede permitir que rayos que no intersecan con la raíz de un subárbol de Cut_t lo recorran. Desgraciadamente, los resultados que obtuvimos no fueron exitosos y se abandonó esta técnica.

4.7. Evolución en el Rendimiento

En secciones anteriores se han mostrado trabajos cuyos objetivos son implementar eficientemente los algoritmos de ray tracing. En la tabla 4.4 realizamos una comparación en el rendimiento de algunos de estos trabajos a fin de analizar su evolución. Hemos elegido los millones de rayos trazados por segundo (MRays/s) como unidad de medida. No todos los trabajos exponen sus resultados en esta unidad de medida, en cuyo caso, la hemos inferido a partir de la información original expuesta en esos artículos.

La comparación del rendimiento es muy difícil por las siguientes razones. Primero, el hardware sobre los que se ejecutan es diferente, por lo que una técnica que sea muy efectiva sobre un sistema puede no serlo sobre otro. Segundo, las escenas tridimensionales usadas en los experimentos pueden ser diferentes. Aún en el caso de que sean las mismas, diferentes posiciones de la cámara pueden hacer variar el rendimiento. Tercero, no todos los algoritmos de ray tracing son iguales. Así, por ejemplo, un trabajo puede orientar su implementación a los rayos primarios mientras que otro lo hará a los secundarios. Por último, no solo el rendimiento es importante, sino también la calidad de la imagen renderizada. Así, un trabajo puede acelerar el trazado de los rayos a costa de obtener imágenes menos realistas. Por todo esto, la tabla 4.4 no pretende ser ni exhaustiva ni completa, sino solo mostrar el rendimiento de algunos sistemas para aportar información sobre la evolución de los algoritmos de ray tracing.

A partir de la información de la tabla podemos inferir varios hechos. El primero es que existe una disminución en el rendimiento de aquellos sistemas que han implementado rayos secundarios que no sean de sombra. Esto se puede observar en los trabajos [AL09], [ALK12] y [TMG11]. Como ya hemos señalado, esta disminución en el rendimiento es consecuencia del aumento de la incoherencia de estos rayos, lo que dificulta el aprovechamiento del hardware.

Por otro lado, un recorrido con frustums aumenta en gran medida el rendimiento de los rayos primarios. Esto se puede ver en los trabajos [RSH05] para CPU, y [GL10] para GPU. En este último trabajo también se realizó un recorrido con frustums para los rayos secundarios previamente ordenados. En un recorrido con frustums, el conjunto de rayos interiores a este puede descartar rápidamente un subárbol de la EA con unas pocas operaciones. Si los rayos son geométricamente muy parecidos, la eficiencia de esta técnica aumenta debido a que los frustums son muy estrechos.

También se puede observar que las mayores ganancias en el rendimiento de los algoritmos de ray tracing vienen determinadas por el hardware. Si se pone en relación el hardware usado en cada trabajo con su rendimiento, se aprecia que los mejores resultados se han obtenido en GPUs, en vez de en CPUs. Dentro de la categoría de las GPUs, se observa un salto en el trabajo [AL09], ejecutado sobre una GeForce 285 GTX, con respecto a los anteriores, ejecutados sobre una GeForce 8800. Esta tendencia en el crecimiento del rendimiento se mantiene en nuevas versiones de GPUs (consultar [ALK12] para más detalles).

Artículo	Año	Escena	Resolución	Hardware	Algoritmo	Rendimiento (MRays/s)	Nota
Wald et al. [WBWS01]	2001	ConfRoom	512 × 512	Pentium III 800MHz	RTW	0.414	
Reshetov et al. [RSH05]	2005	ConfRoom	1024 × 1024	Pentium 4 con HT	Primarios	19.5	Con frustums de rayos
Reshetov et al. [RSH05]	2005	ConfRoom	1024 × 1024	Pentium 4 con HT	Primarios + 1 sombra	31.2	Con frustums de rayos
Boulos et al. [BEL+07]	2007	ConfRoom	-	Opteron 880, 2.46GHZ	Primarios	3.25	Sólo 1 core
Boulos et al. [BEL+07]	2007	ConfRoom	-	Opteron 880, 2.46GHZ	RTW	1.79	Sólo 1 core
Boulos et al. [BEL+07]	2007	ConfRoom	-	Opteron 880, 2.46GHZ	RTD	0.88	Sólo 1 core
Overbeck et al. [ORM08]	2008	FairyForest	1024 × 1024	Intel Xeon 2.06 GHZ (8 cores)	RTW + 1 reflex. + 1 refrac.	>20.1	Paquetes de 16 × 16
Carr et al. [CHH02]	2002	Office	-	GeForce 4	RTD con photon mapping	0.109	
Foley y Sugerman [FS05]	2005	Kitchen	512 × 512	ATI X800 XT PE	Primarios	0.29	KD-Tree backtrack
Horn et al. [HSMH07]	2007	ConfRoom	1024 × 1024	ATI Radeon X1900 XTX	Primarios + 1 specular	≈7	Primarios vía raster; resultado sólo specular
Roger et al. [RAH07]	2007	Kitchen	512 × 512	GeForce 8800 GTS	Primarios + 1 sombra + 2 specular	1.48	Primario y sombra vía raster
Günther et al. [GPSS07]	2007	ConfRoom	1024 × 1024	GeForce 8800 GTX	Primarios + 1 sombra	12.2	
Popov et al. [PGSS07]	2007	ConfRoom	1024 × 1024	GeForce 8800 GTX	Primarios	16.7	Paquetes
Popov et al. [PGSS07]	2007	ConfRoom	1024 × 1024	GeForce 8800 GTX	Primarios + 1 sombra + 1 specular	20.1	Paquetes, sólo la mesa es specular
Zhou et al. [ZHWG08]	2008	FairyForest	1024 × 1024	GeForce 8800 ULTRA	Primarios + 2 sombra	38.46	Pila por rayo
Aila y Laine [AL09]	2009	ConfRoom	1024 × 768	GeForce 285 GTX	Primarios	135.6	Persistent while-while
Aila y Laine [AL09]	2009	ConfRoom	1024 × 768	GeForce 285 GTX	PT, rutas de long. 2	62.4	Persistent while-while
Garanzha y Loop [GL10]	2010	ConfRoom	1024 × 768	GeForce 285 GTX	Primarios	147	Frustum
Garanzha y Loop [GL10]	2010	ConfRoom	1024 × 768	GeForce 285 GTX	Primarios + sombra	69	Frustum
Aila et al. [ALK12]	2012	ConfRoom	1024 × 768	GeForce 480 GTX	Primarios	272.1	Speculative Persistent while-while
Aila et al. [ALK12]	2012	ConfRoom	1024 × 768	GeForce 480 GTX	PT, rutas de long. 2	126.1	Speculative Persistent while-while
Torres et al. [TMG09a]	2009	ConfRoom	1024 × 1024	GeForce 280 GTX	Primarios	34.52	`packet-warp`
Torres et al. [TMG11]	2011	ConfRoom	1024 × 1024	GeForce 285 GTX	Primarios	100.28	Aplicación completa
Torres et al. [TMG11]	2011	ConfRoom	1024 × 1024	GeForce 285 GTX	PT, rutas de long. 2	43.12	Aplicación completa
Torres et al. [TMGA12]	2012	ConfRoom	1024 × 1024	GeForce 285 GTX	Primarios	157.52	SPHERE-OBLI
Torres et al. [TMGA12]	2012	ConfRoom	1024 × 1024	GeForce 285 GTX	PT, rutas de long. 2	46.92	SPHERE-OBLI + CUBE-ORTH

Tabla 4.4: Comparación del rendimiento de los sistemas de ray tracing propuestos en distintos trabajos. Se han separado en tres grupos. El primero corresponde a los trabajos que se ejecutan en CPU. El segundo, los que se ejecutan sobre GPU. El tercero contiene los resultados de nuestros trabajos: el recorrido de una BVH hilvanada (sección 4.5), el recorrido de una BVH mediante un corte (sección 4.6), y el recorrido de tres KD-Trees especializados (sección 3.9). Dentro de un mismo grupo, los trabajos están ordenados por fecha de publicación.

Capítulo 5

Generación de Rutas Coherentes

5.1. Introducción

En el capítulo 4 se subrayó la importancia de agrupar los rayos en grupos coherentes antes de ejecutar su recorrido por la EA, ya se trate de un recorrido mediante paquetes de rayos o con frustums, o con una exploración por rayo individual. En cualquiera de estas situaciones, la coherencia supone un mejor aprovechamiento de los recursos del hardware durante la fase de recorrido.

Para aprovechar la coherencia se pueden seguir dos enfoques. El primero consiste en generar un lote de rayos y posteriormente agruparlos para aprovechar la coherencia existente entre ellos. Esta agrupación se puede realizar antes del recorrido (agrupamiento a priori) o durante el propio recorrido (agrupamiento por comportamiento), y posiblemente usando criterios heurísticos. A veces, la agrupación es eficaz y poco costosa, como sucede para los rayos primarios o para los secundarios por reflexión perfecta (Boulos et al. [BEL+07]). Sin embargo, en otros casos es más exigente y es necesario usar algoritmos de ordenación (Garanzha y Loop [GL10]) o de clasificación (Torres et al. [TMG11]), que añaden un coste extra significativo al rendimiento global del sistema.

El segundo enfoque consiste, por el contrario, en generar grupos de rayos que ya sean coherentes. Esta *generación de rayos coherentes* (o *GRC*) tiene la ventaja de que no es necesaria ninguna reagrupación posterior. Sin embargo, la coherencia de un grupo de rayos solo se puede determinar durante su recorrido a través de la EA, por lo que su generación, que es previa al recorrido, tiene que usar criterios heurísticos a priori que permitan generar estos rayos.

Modificar la forma en que se generan las rutas aleatorias implica que se pueden introducir dependencias entre ellas. Esto supone dos inconvenientes. Por un lado, el ruido típico de los algoritmos de renderizado basados en Monte Carlo se convierte en *ruido estructurado* (sección 5.7). Este ruido estructurado se presenta en la imagen en forma de un patrón regular que es molesto para un adecuado reconocimiento de la imagen renderizada. Para reducir el efecto del patrón se requiere habitualmente un mayor número de muestras para que ese patrón deje de ser visualmente apreciable.

El otro inconveniente consiste en que esas dependencias entre rutas pueden aumentar la diferencia entre las imágenes obtenidas con respecto a aquellas obtenidas usando el mismo número de muestras, pero siendo estas independientes. Sin embargo, las rutas generadas de forma coherentes requieren menores tiempos de recorrido de la EA. Esto supone que, aunque esta diferencia, o error, de estas rutas dependientes sea mayor, más rutas terminarán en el mismo tiempo, haciendo disminuir el error de la imagen.

5.2. Trabajos Relacionados

Gran parte de los trabajos relacionados con la generación coherente de rayos se han presentado en el contexto de *radiosity*. Sbert et al. [SMP98] proponen un nuevo algoritmo para resolver el problema de radiosity. Primero, los objetos de la escena se dividen en secciones, llamadas *patches*. Después, se genera aleatoriamente una recta que atraviesa completamente la escena, y se calculan todos los puntos de intersección entre esa recta y los objetos de la escena. De esa manera, se determina la visibilidad entre varias parejas de patches, procediéndose entonces a calcular el intercambio de energía entre ellas. Los autores llaman a este método *global line Monte Carlo*, ya que en cada iteración se usa una única línea recta en la evaluación de la visibilidad entre patches.

Para aumentar la eficiencia de la técnica anterior, se realiza un preproceso llamado *first shot*, consistente en que los patches emisores de luz difunden primero su energía a sus patches directamente visibles. Estos patches reflejarán posteriormente parte de esa energía, convirtiéndose, a su vez, en nuevas fuentes de luz. Esto permite que la línea global usada encuentre más patches emisores a su paso, facilitando la resolución de radiosity.

Castro et al. [CSN04] incorporan al algoritmo anterior una representación jerárquica de la escena, considerando cada objeto como una caja. Así, la información de energía sobre los patches de la superficie de las cajas se obtiene recursivamente mediante global line Monte Carlo. Esta estructuración de la escena permite disminuir el número total de intersecciones de la línea global al realizarse jerárquicamente. Martínez et al. [MFM09] amplían el algoritmo global line Monte Carlo para introducirle paralelismo mediante el uso de un clúster de PCs. En concreto, las etapas first shot y el cálculo de las intersecciones de cada línea se realiza en paralelo sobre PCs diferentes.

Szirmay-Kalos y Purgathofer [SKP98] resuelven el problema de radiosity usando caminos aleatorios, pero considerando lotes de rayos con la misma dirección, llamados *global ray-bundle*, en cada iteración. Ya que estos conjuntos de rayos tienen la misma dirección, la tubería gráfica se puede usar para obtener el punto de intersección más cercano de cada rayo. Así, se colocan dos planos perpendiculares a la dirección global de los rayos de manera que separen los patches emisores de los receptores. Posteriormente, cada patch se proyecta ortogonalmente sobre uno de los dos planos, usando el hardware gráfico. El resultado es que cada patch receptor conoce los índices de sus patches emisores visibles en la dirección global.

Hermes et al. [HHGM10] discretizan la escena en lo que llaman un *atlas de texturas*. Así, solo se tiene en cuenta el intercambio de energía entre téxeles de ese atlas, de manera similar al intercambio de energía de los patches en radiosity. Al igual que Hachisuka [Hac05], se eligen direcciones aleatorias y la escena se renderiza completamente, situando el plano de proyección fuera de la escena. En vez de usar la técnica de depth-peeling, usan un k-buffer, permitiendo obtener varios valores de profundidad en una sola pasada. Esos puntos se usan para calcular el intercambio de energía, cuyo resultado se guarda de nuevo en el atlas de texturas.

A. Keller [Kel97] discretiza la emisión de las luces sin la necesidad de dividir la escena en patches. Esa discretización consiste en trazar varias rutas aleatorias desde las luminarias y situar nuevas luces puntuales (llamadas *luces puntuales virtuales* o *VLP*) en los lugares donde intersecan con la escena. Debido a que cada VLP es una luz puntual, la tubería gráfica se puede usar para obtener la imagen final. Esto se lleva a cabo renderizando la escena varias veces, considerando cada VLP como una fuente de luz puntual y acumulando los resultados obtenidos para cada una de ellas.

Tokuyoshi y Ogaki [TO12] usan ray-bundles combinados con VLPs para implementar un Bidirectional Path Tracing (BPT) con la tubería gráfica. Una vez obtenidos los puntos de intersección más cercanos de los rayos primarios, se selecciona una dirección global y se proyecta ortogonalmente la escena entera en esa dirección, resultando en una lista de puntos para cada fragmento. Los puntos de intersección de los rayos secundarios se obtienen consultando la lista de cada fragmento. La radiancia de los rayos secundarios se obtiene a través de los *shadow maps*

de cada VLP. El resultado de combinar estas dos técnicas supone el procesamiento de rutas de longitud 3 con la tubería gráfica.

Sadaghi et al. [SCJ09] implementan un Path Tracing (PT) en CPU, al estilo del presentado por Wald et al. [WBS07], en el que todos los rayos de un paquete usan los mismos números aleatorios. Esto implica que los rayos secundarios serán muy parecidos en sucesivos rebotes, aunque con el inconveniente de introducir ruido estructurado en la imagen. El uso de *Interleaved Sampling* (Keller y Heidrich [KH01]) reduce el efecto molesto de este patrón, sobre todo en las imágenes intermedias.

Segovia et al. [SIP07] incorporan una nueva estrategia de mutación a las rutas en el algoritmo de Metropolis Light Transport (MLT). En concreto, la estrategia en la que solo una ruta candidata es evaluada, procedimiento inherentemente secuencial, es sustituida por otra en la que se genera una serie de rutas candidatas, pudiéndo ser estas evaluadas en paralelo. Esta estrategia se usa solo en la mutación de la subruta que parte de la cámara, lo que permite aprovechar la coherencia de los rayos primarios.

T. Hachisuka [Hac04, Hac05] aprovecha el hardware gráfico para realizar un *final gathering* sobre un mapa de fotones en GPU para escenas cuyos materiales son diffuse. Primeramente, los fotones son trazados sobre la escena, usando la CPU. En los vértices de los triángulos se calcula el valor de irradiancia total mediante consultas sobre el mapa de fotones. Posteriormente, en GPU se calculan los puntos de intersección de los rayos primarios, lo que se consigue rasterizando la escena con proyección perspectiva y obteniendo estos puntos a partir de los fragmentos resultantes.

Para realizar final gathering, se elige una dirección global que se usa para proyectar ortogonalmente la escena. El renderizado se realiza en varias fases, usando la técnica de *depth-peeling* de C. Everitt [Eve01] para obtener fragmentos ordenados en profundidad. Con esto se consigue que cada punto de shading obtenga su punto de intersección más cercano. La irradiancia en esos puntos se obtiene interpolando los valores de irradiancia de los vértices, calculados con photon mapping en la etapa anterior.

Novák et al. [NHD10] proponen generar una nueva ruta siempre que otra ruta haya terminado (regeneración de rutas). Este enfoque permite que un hilo que estaba asignado a una ruta comience una nueva, evitando pérdida de recursos en las GPUs. Además, los autores implementan un BPT en el que todas las subrutas de la cámara trazan rayos de sombra a una única subruta de luz, aumentando, por lo tanto, la coherencia de estos rayos.

D. Antwerpen [Ant11] propone combinar la regeneración de rutas con un procesamiento en flujo de los rayos para que no descienda la eficiencia de los algoritmos de ray tracing en GPU. En este sentido, propone algoritmos eficientes de Path Tracing, Bidirectional Path Tracing y Metropolis Light Transport.

5.3. Ecuación de la Cámara

El cálculo del color de un píxel de la imagen final se obtiene sumando la contribución de todas las rutas que aportan energía a ese píxel. Una manera alternativa de obtener ese valor se presentó en la sección 1.5, y consiste en la suma de la contribución de todas las rutas que llegan a la película de la cámara multiplicada por la función de importancia del propio píxel. Matemáticamente, esto se expresa como la integral de la ecuación 1.11, que reproducimos nuevamente a continuación:

$$I_j = \int_{\pi \in \Omega} W_e^{(j)}(\pi)\, f(\pi)\, d\mu(\pi) \tag{5.1}$$

donde I_j es el valor del píxel, Ω es el conjunto de todas las rutas, f es la contribución de una ruta y $W_e^{(j)}$ es la función de importancia del píxel j-ésimo. La función $W_e^{(j)}$ suele ser cero en todo el plano de la película excepto en una región próxima al píxel.

Obsérvese que las ecuaciones de todos los píxeles tienen en común que dependen de la contribución de todas las rutas que llegan a la cámara. La integral sobre todas estas rutas recibe el nombre de *ecuación de la cámara*

$$\int_{\pi\in\Omega} f(\pi)\, d\mu(\pi) \tag{5.2}$$

Esta integral representa la energía total que llega a la cámara desde todas las fuentes de luz. Veremos a continuación que la resolución de la ecuación 5.2 servirá para resolver las ecuaciones 5.1 de todos los píxeles mediante un algoritmo de dos fases.

En la primera fase, se resuelve la ecuación de la cámara 5.2 por Monte Carlo, es decir, trazando rutas aleatorias desde la cámara hasta las luces. La media aritmética de las contribuciones de estas rutas dividido por sus pdfs da como resultado una aproximación a la ecuación de la cámara. Sin embargo, nuestro objetivo no es resolver la ecuación de la cámara, sino resolver las ecuaciones 5.1 de todos los píxeles. Esto se consigue usando estas muestras *globales* como muestras *locales* para todas las ecuaciones de los píxeles. En otras palabras, las ecuaciones de todos los píxeles tendrán en común las mismas muestras, aquellas obtenidas para resolver la ecuación de la cámara. Así, en la segunda fase del algoritmo, la ecuación de un píxel se aproxima aplicando su función de importancia a todas las muestras obtenidas en la fase anterior y calculando su media aritmética.

En nuestro modelo de partículas (sección 1.5), las funciones de importancia son constantes en todo el recuadro del píxel y cero fuera de él. Por tanto, no es posible que una misma muestra aporte energía a dos píxeles diferentes. Sin embargo, considerar que las rutas que contribuyen a píxeles diferentes son muestras que resuelven una misma ecuación permite aplicar las mismas técnicas de resolución de una integral (la ecuación de la cámara 5.2) a un conjunto de integrales (las ecuaciones de renderizado 5.1 de cada píxel), como se verá en las siguientes secciones.

5.4. Generación de Rutas

Supongamos que se ha encontrado el punto de intersección más cercano de un rayo y hay que elegir la dirección del siguiente rayo de la ruta. Esa dirección se obtiene eligiendo aleatoriamente un punto ω sobre la superficie del hemisferio $\mathcal{H}_p$, situado sobre el punto de intersección p. La pdf del punto ω viene determinada por la BRDF del objeto en el punto p. La forma de realizar esto consiste en elegir primero un punto $\omega^{(L)}$ sobre el hemisferio canónico $\mathcal{H}$ (coordenadas locales) y transformarlo posteriormente a otro punto ω a coordenadas mundiales.

Por convenio, el hemisferio canónico $\mathcal{H}$ está definido sobre la base canónica $\{(1,0,0), (0,1,0), (0,0,1)\}$, de manera que el vector $(0,0,1)$ apunta al polo del hemisferio (figura 5.1a). El hemisferio $\mathcal{H}_p$ en coordenadas mundiales se sitúa sobre el punto de intersección p, de forma que el vector normal N_p en ese punto apunta hacia su polo (figura 5.1b).

Para transformar las coordenadas de un punto de locales a mundiales es necesario establecer un *espacio tangente* sobre el punto de intersección del rayo. Un espacio tangente está definido por tres vectores $[U_p, V_p, N_p]$ que forman una base ortonormal, donde el tercero de ellos corresponde con la normal N_p de la superficie en el punto p. Aunque existen infinitos espacios tangentes con la característica anterior para un cierto punto p, solo consideraremos el espacio tangente que se construye de la siguiente forma. Supongamos la existencia de un vector, llamado *up*, cuyo único requisito es que no sea paralelo a la normal N_p. A partir de estos dos vectores se definen los otros dos vectores del espacio tangente como

$$U_p = \frac{up \times N_p}{||up \times N_p||} \qquad \text{y} \qquad V_p = \frac{N_p \times U_p}{||N_p \times U_p||}$$

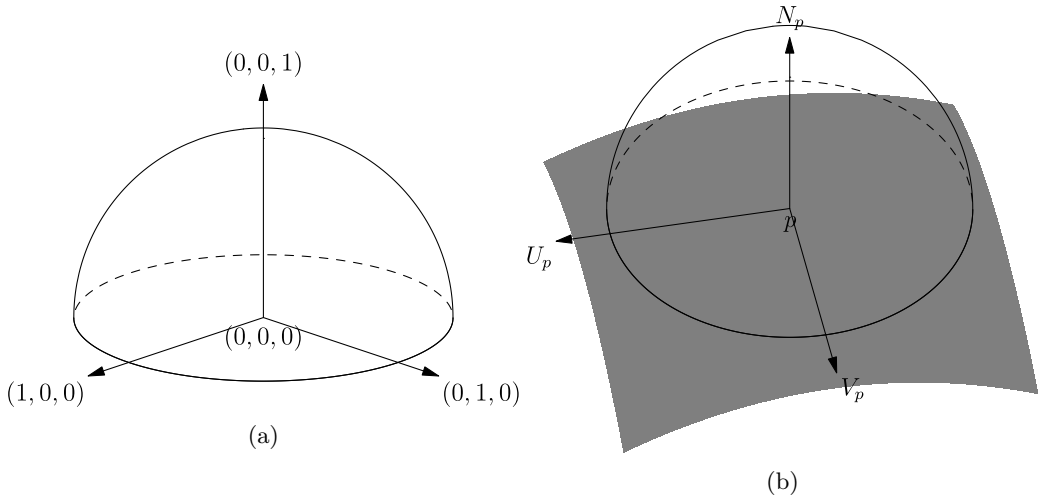


Figura 5.1: Fig. (a). Hemisferio canónico ($\mathcal{H}$). El vector $(0, 0, 1)$ señala a su polo. Fig (b). Hemisferio sobre un punto p de la escena ($\mathcal{H}_p$). En este hemisferio, el vector normal a la superficie N_p apunta al polo.

Una vez obtenido el espacio tangente, las coordenadas globales ω del vector en coordenadas locales $\omega^{(L)} = (\omega_x^{(L)}, \omega_y^{(L)}, \omega_z^{(L)})$ se obtienen como

$$\omega \quad = \quad \omega_x^{(L)} \cdot U_p \; + \; \omega_y^{(L)} \cdot V_p \; + \; \omega_z^{(L)} \cdot N_p$$

Supongamos ahora que el vector *up* es común a todas las rutas. Si las normales sobre los puntos de intersección de dos rayos son parecidas, entonces sus espacios tangentes, ET_1 y ET_2, serán también muy parecidos. Si transformamos el vector $\omega^{(L)}$ a coordenadas globales usando los dos espacios tangentes anteriores, entonces obtendremos dos vectores en coordenadas globales, ω_1 y ω_2. Ya que estos espacios tangentes, ET_1 y ET_2, son muy parecidos, los vectores ω_1 y ω_2 serán también muy parecidos (figura 5.2). Este hecho se usará para generar rutas completas cuyos rayos sean parecidos, es decir, la generación de rutas coherentes se basa en la generación de direcciones semejantes tras cada rebote.

Para implementar la generación de rutas coherentes, vamos a cambiar la generación de direcciones en cada punto de intersección de las rutas. El algoritmo de selección de la siguiente dirección consta de dos fases. En la primera, el hemisferio canónico se divide en secciones disjuntas y se elige una de ellas aleatoriamente. Posteriormente, cada ruta elige un punto sobre la sección anteriormente elegida (sección 5.4.1). Si durante la extensión de un conjunto de rutas, todas ellas eligen su siguiente dirección sobre la misma sección del hemisferio, entonces la coherencia del grupo se mantendrá, con mucha probabilidad, en cada rebote (sección 5.4.2).

5.4.1. Generación de Direcciones sobre una Sección Esférica

Consideremos la función T que transforma coordenadas esféricas en cartesianas

$$T(\theta, \phi) = \big(\sin(\theta)\cos(\phi),\ \sin(\theta)\sin(\phi),\ \cos(\theta)\big)$$

donde $\theta \in [0, \frac{\pi}{2}]$ y $\phi \in [0, 2\pi]$. Si dividimos el rango de la coordenada θ en Q partes y el de la coordenada ϕ en M, el hemisferio canónico queda dividido en $Q \cdot M$ secciones esféricas (figura 5.3).

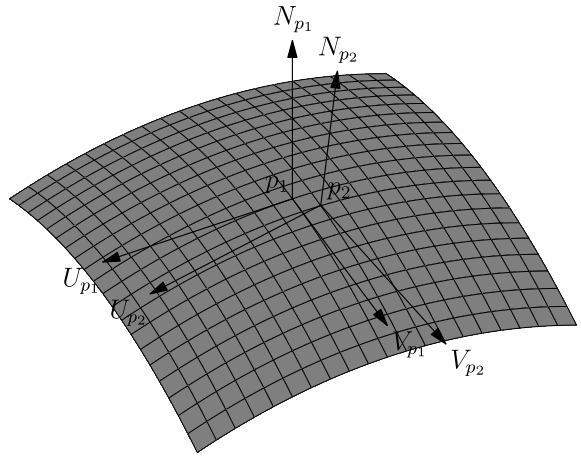


Figura 5.2: Dos espacios tangentes colocados sobre un mismo objeto y en puntos próximos. Se ha usado el mismo vector *up* para construir los espacios tangentes.

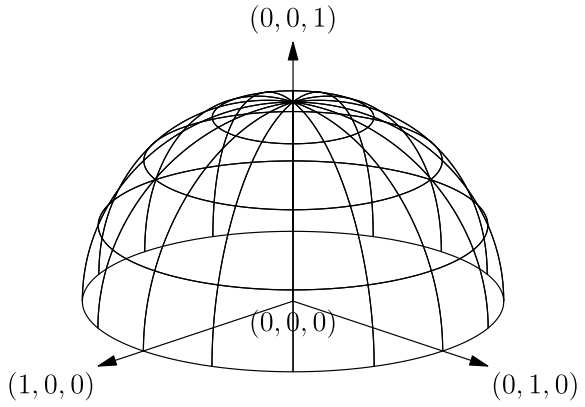


Figura 5.3: Secciones esféricas del hemisferio canónico. La coordenada esférica θ se ha dividido en 4 secciones y la coordenada esférica ϕ, en 16.

Cada sección SP_{ij} queda entonces definida como

$$SP_{ij} = \Big\{ T(\theta,\phi) \quad | \quad \theta \in [i\Delta\theta, (i+1)\Delta\theta], \quad \phi \in [j\Delta\phi, (j+1)\Delta\phi] \Big\}$$

donde $\Delta\theta = \frac{\pi/2}{Q}$, $\Delta\phi = \frac{2\pi}{M}$, $i \in \{0, 1, \ldots, Q-1\}$ y $j \in \{0, 1, \ldots, M-1\}$.

Supongamos que el hemisferio canónico se encuentra dividido en secciones esféricas. La generación de direcciones se realiza como sigue. En primer lugar, se elige aleatoriamente una sección esférica de entre las $Q \cdot M$ posibles. Posteriormente, las direcciones en coordenadas locales se eligen aleatoriamente como puntos sobre esa sección esférica. Finalmente, cada dirección se transforma a coordenadas mundiales sobre el espacio tangente del punto de intersección.

Para que la efectividad del método de Monte Carlo sea alta, la pdf de muestreo del hemisferio debe ser parecida a la BRDF de la superficie, lo que se conoce como *important sampling* (sección 1.6.7). Las superficies diffuse son las que generan rayos más incoherentes ya que la siguiente dirección se toma aleatoriamente sobre todo el hemisferio. Por lo tanto, a lo largo de este capítulo, solo vamos a considerar la generación de direcciones en los puntos de las rutas situados sobre

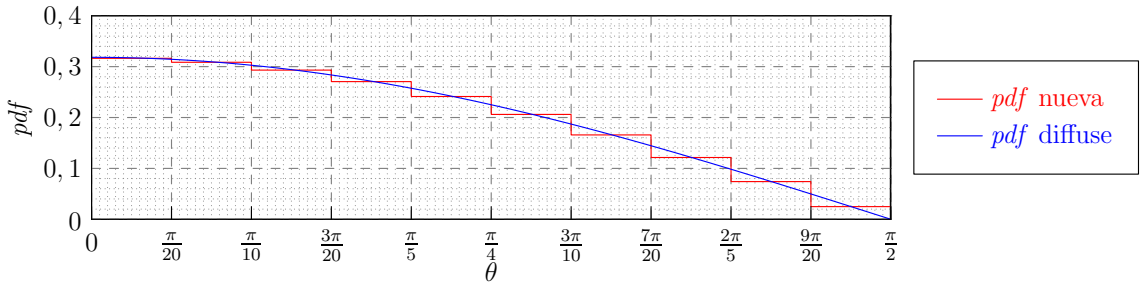


Figura 5.4: Comparación entre la pdf de la generación de direcciones sobre una sección esférica (curva roja) con la pdf de la BRDF diffuse (curva azul). En el eje x se encuentra el ángulo θ y en el eje y se encuentra el valor de la pdf. El número de secciones en las que se ha dividido la coordenada θ es 10.

superficies diffuse. Habitualmente, el muestreo de un hemisferio para una superficie diffuse se lleva a cabo con una pdf proporcional al coseno del ángulo que se forma con la normal (sección 1.7.2). Por tanto, para que la pdf obtenida en la generación de direcciones sobre una sección esférica sea también similar al coseno de este ángulo, tenemos que establecer adecuadamente la probabilidad de elección de una sección esférica y la pdf de elección de un punto dentro de la sección elegida. El procedimiento es como sigue. Primero, se genera un punto o sobre el hemisferio con la misma densidad que la usada para la BRDF diffuse, es decir,

$$p(o) = \frac{|N \cdot o|}{\pi}$$

donde $N = (0, 0, 1)$ apunta al polo del hemisferio canónico. La sección SP_{ij} elegida será aquella sobre la que caiga el punto o. Así, la probabilidad $P(i, j)$ de que la sección esférica SP_{ij} sea elegida es

$$P(i,j) = \int_{o \in SP_{ij}} p(o)\, d\sigma(o)$$

Ya que el hemisferio en coordenadas locales tiene el vector $(0, 0, 1)$ como polo, entonces la probabilidad de elección de la sección esférica SP_{ij} es

$$P(i,j) = \int_{j\Delta\phi}^{(j+1)\Delta\phi} \int_{i\Delta\theta}^{(i+1)\Delta\theta} \frac{\cos(\theta)}{\pi} \sin(\theta)\, d\theta\, d\phi = \frac{\Delta\phi}{2\pi} \left(\cos^2(i\Delta\theta) - \cos^2((i+1)\Delta\theta)\right)$$

Posteriormente, se tiene que elegir un punto aleatorio dentro de la sección esférica. Esa elección se realiza con probabilidad uniforme dentro de los puntos de la propia sección. Sea $p((i, j) \to \omega)$ la pdf de elección de un punto dentro del SP_{ij} anteriormente seleccionado, entonces

$$p((i,j) \to \omega) = \frac{1}{\sigma(SP_{ij})}$$

donde $\sigma(SP_{ij})$ es el área de la superficie de SP_{ij}

$$\sigma(SP_{ij}) = \int_{\omega \in SP_{ij}} 1\, d\sigma(\omega) = \int_{j\Delta\phi}^{(j+1)\Delta\phi} \int_{i\Delta\theta}^{(i+1)\Delta\theta} \sin(\theta)\, d\theta\, d\phi = \Delta\phi\, (\cos(i\Delta\theta) - \cos((i+1)\Delta\theta))$$

Finalmente, la densidad $p(\omega)$ correspondiente a elegir un punto ω en el hemisferio es el producto de la probabilidad de elección de una sección esférica y de la pdf de selección de un punto sobre dicha sección

$$p(\omega) = P(i,j) \cdot p((i,j) \to \omega) = \frac{1}{2\pi}(\cos(i\Delta\theta) + \cos((i+1)\Delta\theta))$$

Como se puede ver en la figura 5.4, la pdf de la generación de direcciones coherentes se aproxima bastante a la pdf de la BRDF diffuse, y esta aproximación va siendo mejor a medida que Q crece.

5.4.2. Generación de Rutas Coherentes

La generación de direcciones sobre una sección esférica garantiza en gran medida la generación de rutas cuyos rayos son coherentes. Así, consideremos un conjunto de G rayos primarios generados sobre una rejilla de píxeles. Estos rayos ya son geométricamente muy parecidos, por lo que es muy probable que sean muy coherentes durante el recorrido y que sus puntos de intersección se encuentren muy próximos. En esos puntos, los G rayos implicados eligen una dirección aleatoria para extender sus rutas. Si el vector *up* y la sección esférica es común a todos los G rayos, entonces todas las direcciones generadas serán muy parecidas, por lo que los siguientes rayos de las rutas serán nuevamente similares y, por tanto, coherentes con mucha probabilidad. Así, esta nueva forma de generar direcciones facilita que la coherencia de los rayos primarios se mantenga tras cada rebote.

Obsérvese que no existe completa garantía de que las rutas generadas sean coherentes. Así, es posible, por ejemplo, que dos puntos de intersección se encuentren cerca, pero que sus normales sean muy diferentes. En este caso, aunque el vector *up* y la sección esférica sean los mismos en ambos puntos, los espacios tangentes serán muy diferentes y la coherencia no se mantendrá. Además de la geometría de la escena, la terminación de rutas y el tamaño de las secciones esféricas influyen en la pérdida de coherencia de las rutas. Esto se verá con más detalle en la sección 5.5.

Las direcciones generadas sobre una misma sección esférica no son independientes, por lo que las rutas a las que pertenecen tampoco lo son. Esto se puede demostrar comprobando que la densidad de probabilidad conjunta de dos direcciones generadas sobre la misma sección no es igual al producto de las marginales. Así, sean ω_1 y ω_2 dos direcciones elegidas sobre la misma sección SP_{ij}. Su pdf conjunta es

$$p(\omega_1, \omega_2) = P(i,j) \cdot p((i,j) \to \omega_1) \cdot p((i,j) \to \omega_2)$$

mientras que el producto de las densidades marginales es

$$p(\omega_1)p(\omega_2) = P^2(i,j) \cdot p((i,j) \to \omega_1) \cdot p((i,j) \to \omega_2)$$

por lo que se cumple $p(\omega_1, \omega_2) \neq p(\omega_1)p(\omega_2)$, demostrando que ambas direcciones no son independientes.

5.5. Ruleta Rusa por Grupo

Los grupos coherentes se determinan por la posición que ocupan los rayos en el array de rayos. Si G es el número de rayos de cada grupo, entonces cada grupo de G rayos consecutivos en el array constituye un grupo. Durante el renderizado, es posible que alguna ruta termine antes que el resto de rutas de su mismo grupo. Existen dos maneras de gestionar esta terminación. La primera consiste en no generar nuevas rutas hasta que todo el lote de rutas haya terminado. Este procedimiento tiene el inconveniente de que el grado de paralelismo disminuye a medida que el renderizado avanza, ya que las rutas van terminando. Como consecuencia, esta solución no es adecuada para las GPUs en la práctica.

El otro enfoque consiste en generar una nueva ruta cada vez que alguna termina, manteniéndose constante el tamaño del lote de rayos. Este método recibe el nombre de *regeneración* de rutas. La cuestión que surge entonces tiene que ver con cómo rellenar el hueco dejado por la ruta que ha terminado. Si el rayo de la nueva ruta ocupa el lugar dejado por la ruta terminada entonces el grupo pasará a estar formado por rayos que se pueden encontrar en cualquier parte de la escena junto con el rayo primario de la ruta regenerada, disminuyendo la coherencia del grupo. Si el hueco dejado por la ruta terminada se ocupa por otro rayo de otro grupo, entonces también disminuye la coherencia ya que la secuencia de secciones esféricas usadas no es la misma para todas las rutas del nuevo grupo. En definitiva, es importante evitar que un grupo coherente se rompa debido a que sus rutas terminan en distintos momentos.

Una ruta puede terminar por tres causas: se sale de la escena, llega a una luz, o no pasa el test de la ruleta rusa. Las dos primeras causas son debidas a la geometría de la escena, por lo que la generación de rutas coherentes tiende a evitar, por sí misma, la ruptura de un grupo coherente. La tercera causa, la ruleta rusa, es el caso más probable de pérdida de coherencia dentro de un grupo, ya que es difícil que todas las rutas superen este test a la vez. Por tanto, es necesario modificar la ruleta rusa para mantener la coherencia de un grupo de rayos durante más tiempo.

La razón por la que la ruleta rusa puede romper un grupo de rayos coherentes es porque se realiza de manera individual. Una forma de evitar el problema consiste en ejecutar este test por grupo. Esta técnica, que llamaremos *ruleta rusa por grupo* (o *RRG*) se basa en terminar probabilísticamente todas las rutas del mismo grupo a la vez. Desde un punto de vista matemático, la RRG consiste en sustituir la media aritmética de las rutas de un grupo por cero, de manera probabilística. Así, sean $\pi_1, \cdots, \pi_G$ las G rutas aleatorias de un grupo. A partir de las rutas anteriores se pueden obtener G estimadores unbiased de la ecuación de la cámara (ecuación 5.2) si se divide la contribución de cada ruta por su pdf, $\hat{I}_i = \frac{f(\pi_i)}{p(\pi_i)}$. De esta forma, su media aritmética

$$\hat{F}_G = \frac{1}{G} \sum_{i=1}^{G} \hat{I}_i$$

es también un estimador unbiased de la ecuación de la cámara. Este estimador $\hat{F}_G$ puede sustituirse por el estimador Z, definido como

$$Z = \begin{cases} \frac{\hat{F}_G}{q} & \text{con probabilidad } q \\ 0 & \text{con probabilidad } 1 - q \end{cases}$$

donde $q \neq 0$ es la probabilidad de pasar el test de la ruleta rusa. Ya que ambos estimadores tienen la misma esperanza, Z también es una aproximación de la ecuación de la cámara.

El valor final de Z depende de la probabilidad q, por lo que aparecen dos casos: si Z no pasa el test de la ruleta rusa, entonces su valor es 0; si Z pasa el test entonces el valor de $\hat{F}_G$ se divide por q. Sin embargo, no queremos perder la información de cada uno de los G estimadores $\hat{I}_i$, ya que serán usados para resolver las ecuaciones de los píxeles (sección 5.3). Por tanto, tenemos que buscar un procedimiento para obtener los mismos resultados que Z, pero modificando individualmente cada estimador. El procedimiento es el siguiente: si Z no pasa el test de la ruleta ruta, entonces sustituimos todos los estimadores $\hat{I}_i$ por cero; si Z pasa el test, dividimos cada estimador por q. Se puede comprobar fácilmente que la media de estos nuevos G estimadores coincide con el valor

para Z

$$Z = \begin{cases} \frac{1}{G}\sum_{i=0}^{G}\frac{\hat{I}_i}{q} & \text{con probabilidad } q \\ \\ \frac{1}{G}\sum_{i=0}^{G}0 & \text{con probabilidad } 1-q \end{cases}$$

De esta manera, disponemos de un procedimiento para que un conjunto de rutas obtenga el mismo resultado en la ruleta rusa, es decir, la ruleta rusa por grupo.

5.6. Detalles de Implementación

Hemos implementado la generación de rutas coherentes y la ruleta rusa por grupo en [TMG12]. El algoritmo de renderizado que hemos usado es un path tracing (PT) implícito, implementado en CUDA, que explora una BVH construida con SAH. El PT es *iterativo*, de manera que las rutas se siguen extendiendo y regenerando mientras no se haya alcanzado el tiempo de ejecución máximo previamente fijado. Solo al principio del programa, o cuando la cámara cambia de posición, el algoritmo tiene que comenzar a generar la imagen de nuevo. Este esquema sigue, en líneas generales, el trabajo de D. Antwerpen [Ant11].

El algoritmo completo se ha implementado en CUDA con cuatro kernels, cuyos nombres son: *Extends*, *Compact*, *Traversal* y *Display*. Estos kernels se ejecutan secuencialmente en el orden en que se han citado. Todas las imágenes renderizadas tienen una resolución de 1024×1024, que coincide también con el tamaño del lote de rayos, ya que se mantiene siempre un rayo por píxel.

Las rutas se guardan en dos arrays: `rays` e `idrays`. El array `rays` guarda la información necesaria para mantener el estado de cada ruta, esto es, el origen y la dirección del último rayo más la radiancia acumulada de la ruta. La ubicación de los rayos en este array coincide con el código de Morton del píxel al que pertenecen. Sin embargo, este no es el orden en que los rayos recorren la estructura de aceleración. Este orden está especificado en el array de enteros `idrays`.

Al comienzo del renderizado, el kernel *Extends* lanza un hilo por cada píxel de la imagen. El identificador `id_g` de cada hilo en el grid representa el código de Morton de su píxel. Cada uno de estos hilos genera un rayo primario que sale de la cámara por su píxel. Ese rayo se guarda en `rays[id_g]` y el índice `id_g` se guarda en `idrays[id_g]`. Así, el orden de recorrido de los rayos sí coincide con su posición en `rays` en la primera ejecución del kernel *Traversal*. Sin embargo, durante el renderizado, el array `idrays` va a ser reordenado, por lo que el valor `idrays[id_g]` indicará el índice del rayo manejado por el hilo `id_g`. Además, los hilos del kernel *Extends* inician la imagen escribiendo el color negro en cada píxel y cero en el número de rutas generadas.

Traversal encuentra el punto de intersección más cercano de cada rayo usando el orden marcado por `idrays`. Para este kernel se ha seguido el trabajo de Aila y Laine [AL09] que implementa el recorrido mediante *hilos persistentes*. Una vez encontrado el punto de intersección de cada rayo, *Extends* se encarga de extender la ruta, generando la dirección del siguiente rayo. Si no se usa la generación de rutas coherentes, el siguiente rayo se genera aleatoriamente usando la BRDF sobre todo el hemisferio, como se explicó en la sección 1.4. Los detalles sobre cómo se implementa la GRC se explicarán en la sección 5.6.1. *Extends* también se encarga de regenerar una ruta desde la cámara cuando esta ha terminado, de forman similar a como lo hacen Novak et al. [NHD10]. Concretamente, cuando una ruta termina, el hilo asociado a ella acumula su contribución al píxel de la imagen final. Después, el hilo regenera la ruta, lanzando un nuevo rayo primario desde la cámara a través de su píxel.

Tras la ejecución del kernel *Extends*, el kernel *Compact* agrupa los rayos regenerados al final del array, mientras que los extendidos quedan al principio. De esta manera, se aprovecha la

coherencia de los nuevos rayos primarios. Como ya se ha mencionado, esta reasignación de los rayos no se hace directamente sobre el array de rayos `rays`, sino sobre su array de índices `idrays`. Este kernel está implementado usando el *radixsort* de CUDPP 1.1.1 [HOS+10], tomando solo un bit para ordenar.

Por último, el kernel *Display* muestra la información acumulada en la imagen parcial del path tracing. Su implementación se ha realizado para facilitar la interacción y depuración, y no se ha tenido en cuenta en los resultados (sección 5.7).

5.6.1. Implementación de la Generación de Rutas Coherentes

Un grupo de rayos consiste en G rayos consecutivos según el orden marcado por el array `idrays`. Los tamaños de G que hemos probado están relacionados con la forma en que se agrupan físicamente los hilos en CUDA. Concretamente, hemos considerado grupos con el tamaño de un warp ($G = 32$), de un bloque de hilos ($G \in \{256, 512, 1024\}$) y de todo el grid ($G = 2^{20}$).

Cuando el tamaño de G es un warp, un hilo del grupo se encarga de seleccionar la sección esférica común al grupo y de escribirlo en memoria compartida. Posteriormente, el resto de hilos del warp recogen esa información y cada uno prolonga su ruta hacia un punto aleatorio sobre esa sección esférica. En este caso, no es necesario el uso de barreras debido a la forma en que los warps se ejecutan en GPU. Al programa que realiza la generación coherente de esta manera lo llamamos `warp`. En esta configuración, el tamaño de bloque CUDA es siempre de 256 hilos.

Cuando G es un bloque, un hilo del bloque se encarga de elegir la sección esférica común al grupo y de comunicar la elección al resto de hilos a través de memoria compartida. Hemos usado la notación `block_256`, `block_512` y `block_1024` para referirnos a grupos de tamaño $G = 256$, 512 y 1024, respectivamente. A diferencia de `warp`, esta vez sí es necesaria una sincronización explícita entre los hilos de un bloque.

Cuando G es la rejilla entera, todos los rayos del lote pertenecen a un único grupo. En este caso, la CPU es la encargada de elegir aleatoriamente la sección esférica común a todos los rayos. Así, antes de la ejecución de *Extends*, la CPU selecciona una sección del hemisferio canónico y la envía a la memoria global de la GPU. Durante la ejecución de *Extends*, todos los hilos recogerán esa información y generarán su siguiente dirección aleatoriamente en la superficie de esa sección esférica. El nombre que usamos para referirnos a este programa es `grid`.

5.6.2. Implementación de la Ruleta Rusa por Grupo

El test de la ruleta rusa por grupo se realiza en el kernel *Extends*. Si se pasa este test, todos los rayos de un mismo grupo se extienden; en caso contrario, todos terminan y se regeneran. La probabilidad de pasar el test es la misma que en el caso individual (sección 5.7). En `warp` y `block`, solo un hilo del grupo realiza la ruleta rusa, mientras que en `grid` es la CPU la que decide si terminar todos los rayos.

Con el uso de la RRG, es probable que el número de rayos consecutivos coherentes no cambie. De esta forma, el agrupamiento del kernel *Compact* puede no ser necesario. En nuestro trabajo [TMG12] se muestran los resultados de los experimentos, activando y desactivando este kernel. En la sección 5.7 solo se mostrarán los resultados sin *Compact*. Los algoritmos que incorporan la ruleta rusa por grupo y no realizan *Compact* llevan el sufijo `_rr(NO)`.

5.7. Resultados

Las escenas que hemos usado para los experimentos son ConfRoom, FairyForest, Sponza y Stanford (figura 5.9). Todas las imágenes renderizadas tienen una resolución de 1024×1024,

tomadas mediante una cámara *pinhole*. Los materiales de las tres primeras escenas son todos diffuse. En la escena STANFORD son diffuse solo las paredes, mientras que el Buda tiene un material glossy, el Dragón, especular, y el Conejo, de refracción perfecta. Usaremos el valor de 0.8 como probabilidad de continuación de una ruta en todos los tests de ruleta rusa, tanto por grupo como de manera individual.

Cada tipo de escena sirve para probar la GRC sobre rutas de diferentes características. Las escenas CONFROOM y FAIRYFOREST están muy bien iluminadas, así que la mayoría de las rutas tendrán longitud 2. Para la primera, todo el techo de la habitación es un área de luz, mientras que la segunda está abierta por arriba. Por otra parte, SPONZA tiene un área de luz encima del atrio y la cámara está enfocando un lugar donde gran parte de la iluminación es indirecta. Finalmente, STANFORD tiene un área de luz en la parte superior de la escena y superficies no diffuse. Es decir, en esta escena se mezclan superficies sobre las que se aplica la GRC (las paredes) con otras sobre las que no (el resto de superficies).

Los algoritmos anteriormente descritos se han probado sobre una GeForce GTX 580 (cap. 2.0) con 1.5GB de RAM. Se han probado diferentes valores de Q entre 4 y 1000. Para simplificar los experimentos, se ha establecido $M = 4Q$, lo que divide el espacio de las coordenadas esféricas en regiones cuadradas. Hemos implementado también un Path Tracing clásico en el que cada ruta se prolonga independientemente de las demás. A este algoritmo lo hemos llamado `normal` y se usará como referencia. Todos los experimentos se han llevado a cabo renderizando la escena con un límite de tiempo de 30 segundos para el tiempo acumulado de los kernels *Extends*, *Compact* y *Traversal*, excepto para las imágenes de referencia (figura 5.9), que se han obtenido tras 20 minutos de renderizado con `normal`.

Las gráficas de la figura 5.5 muestran el número de rutas terminadas por píxel (en media). El algoritmo `normal` se muestra como una recta constante ya que su ejecución no depende de Q. En primer lugar, se observa que el número de rutas terminadas por todos los algoritmos que incorporan la GRC superan a `normal`. Se ha comprobado experimentalmente (usando *NVidia Visual Profiler*) que el kernel *Traversal* requiere menos tiempo cuando se usa GRC debido a que las cachés tienen una menor tasa de fallos y a que el número de transferencias desde memoria global es menor. En segundo lugar, según crece Q, las direcciones generadas se hacen más parecidas dentro de cada grupo porque las secciones esféricas son cada vez más pequeñas. En consecuencia, las curvas de nuestros algoritmos son no decrecientes, es decir, el recorrido es más rápido según crece Q.

Los algoritmos que generan a nivel de grid ($G = 2^{20}$) son los que ofrecen mejor rendimiento debido a que su tasa de aciertos de caché es la mayor. Después se encuentran los algoritmos basados en bloque y en warp. Las curvas de los algoritmos que generan a nivel de bloque son muy parecidas. Las curvas correspondientes a los tamaños 256 y 512 discurren juntas, y la asociada al tamaño 1024 queda ligeramente por debajo. El motivo es que con bloques de 1024 hilos, solo un bloque se asigna a cada multiprocesador de la GPU, por lo que las barreras de sincronización suponen una penalización mayor que para los otros tamaños.

A medida que Q crece, el número de rutas terminadas en el mismo tiempo de ejecución aumenta debido a la mayor coherencia de las rutas. Esto debería suponer que las imágenes obtenidas tuvieran menos error. Pero al mismo tiempo, la correlación entre las rutas de los píxeles del mismo grupo también es mayor, lo que implica un mayor error. En las gráficas de la figura 5.6 se analiza el error de las imágenes resultantes con respecto a Q. El error está medido como la raíz del error cuadrado medio (*Root Mean Square* o *RMS*) de cada píxel[1] con respecto a las imágenes de referencia (figura 5.9). Para los primeros valores de Q, las gráficas muestran que el error de las imágenes obtenidas usando GRC tienen más error que `normal`, a pesar de que el número de rutas terminadas es mayor. A medida que Q crece, también lo hace el número de rutas terminadas y

[1] Ya que cada píxel está formado por tres valores de color RGB, cada uno se considera un dato individual. Por tanto, el número total de datos que se usa para calcular el error de una imagen es de 3×1024^2.

el error decrece hasta que el error es menor que el obtenido con `normal`. Nuestra interpretación de este hecho es que tiene mayor influencia la ganancia que aporta un mayor número de rutas terminadas que el error que supone un aumento en la correlación de las muestras, ya que se trata de curvas decrecientes, en general. Los algoritmos basados en grid tienen un comportamiento más imprevisible, como se puede ver en las gráficas de FairyForest y Stanford.

En la figura 5.10 presentamos capturas de las imágenes generadas con `block_1024`. La correlación entre las muestras de los píxeles de un mismo grupo se aprecia visualmente como un patrón en forma de rejilla cuadrada. Lo mismo ocurre para los otros tamaños de grupo en la GRC. Este efecto se agudiza para valores grandes de Q y usando ruleta rusa por grupo. Sin embargo, aunque el RMS de estas imágenes es menor que el del algoritmo `normal`, este patrón resulta molesto desde un punto de vista subjetivo.

Este patrón es consecuencia de que, gran parte del tiempo, un mismo grupo está formado por rutas que contribuyen a píxeles contiguos. Para aliviar este efecto, se ha usado la técnica del *interleaved sampling* [KH01], que consiste en que los rayos de varios grupos intercalen los píxeles a los que contribuyen sus rutas. La implementación de esta técnica es sencilla, simplemente hay que multiplicar las coordenadas del píxel asociado a cada hilo por un entero impar D. Los valores que hemos usado para nuestros experimentos han sido $D = 3$ y $D = 5$.

El inconveniente del interleaved sampling es que los rayos primarios tienen menor coherencia ya que son generados para píxeles más alejados entre sí. Esta disminución en la coherencia se aprecia experimentalmente como una disminución del número de rutas terminadas y, por tanto, de un aumento en el error con respecto a `normal` (figuras 5.7 y 5.8), aunque ese error sigue siendo menor que el originado por `normal`. En las figuras 5.11 y 5.12 se muestran algunas capturas para $D = 3$ y $D = 5$, respectivamente. Como se puede apreciar, las imágenes son más claras ya que no se observa el ruido estructurado en forma de patrón regular.

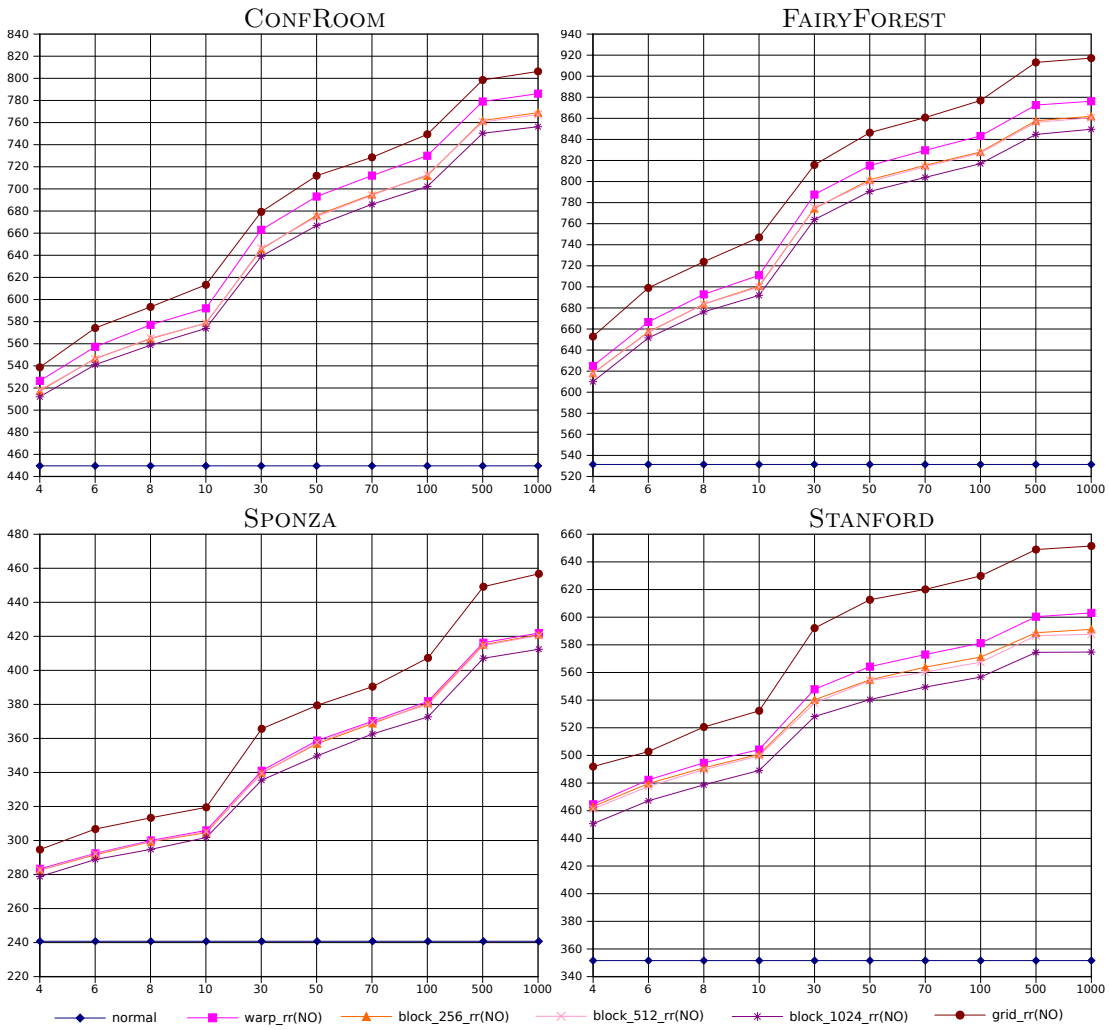


Figura 5.5: Número de rutas terminadas en media (eje y) en relación con Q, número de divisiones de la coordenada θ (eje x). El tiempo de renderizado ha sido de 30 segundos.

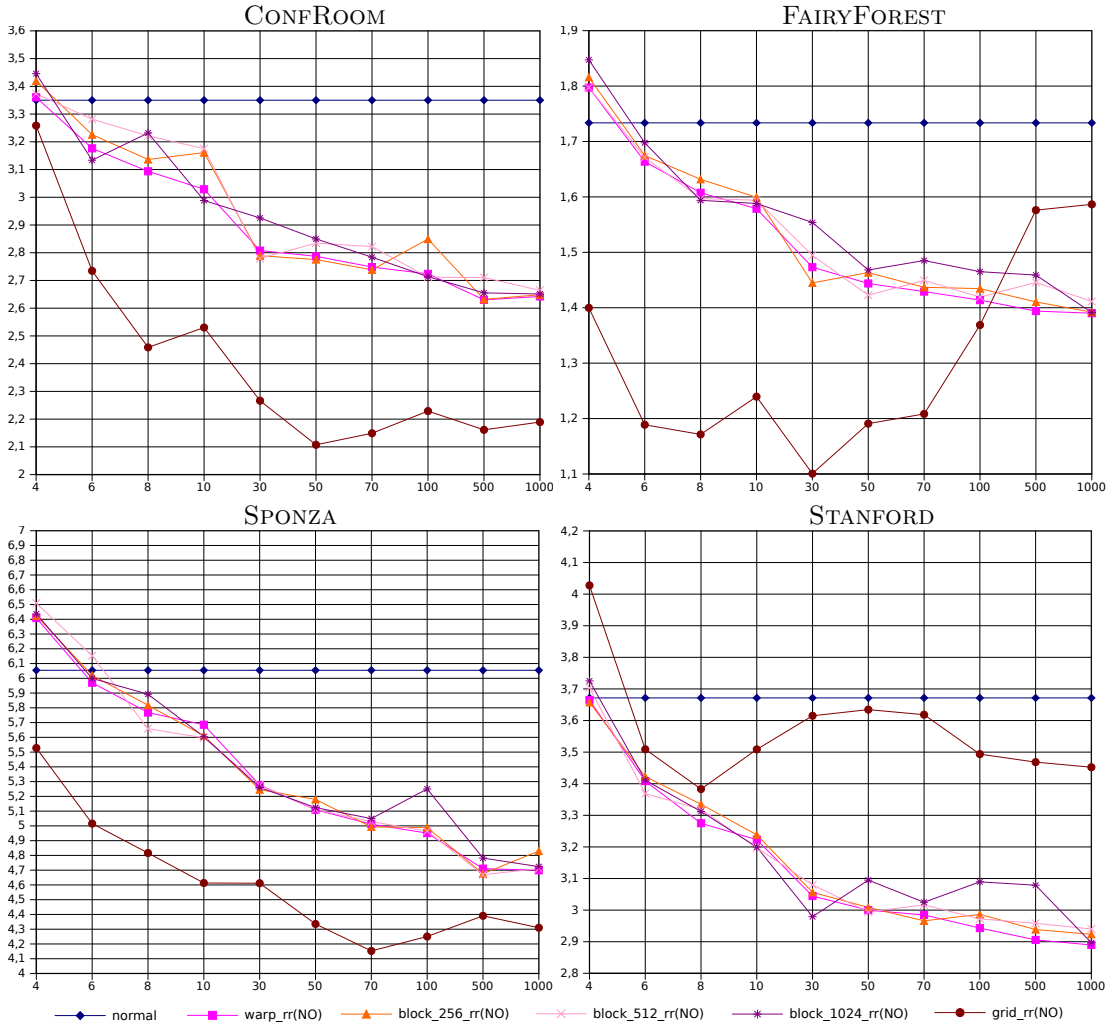


Figura 5.6: Error cuadrado medio (RMS) entre las imágenes renderizadas en 30 segundos con respecto a las imágenes de referencia. En el eje x se encuentra el número de divisiones de θ, es decir, el parámetro Q. En el eje y se muestra el RMS en %.

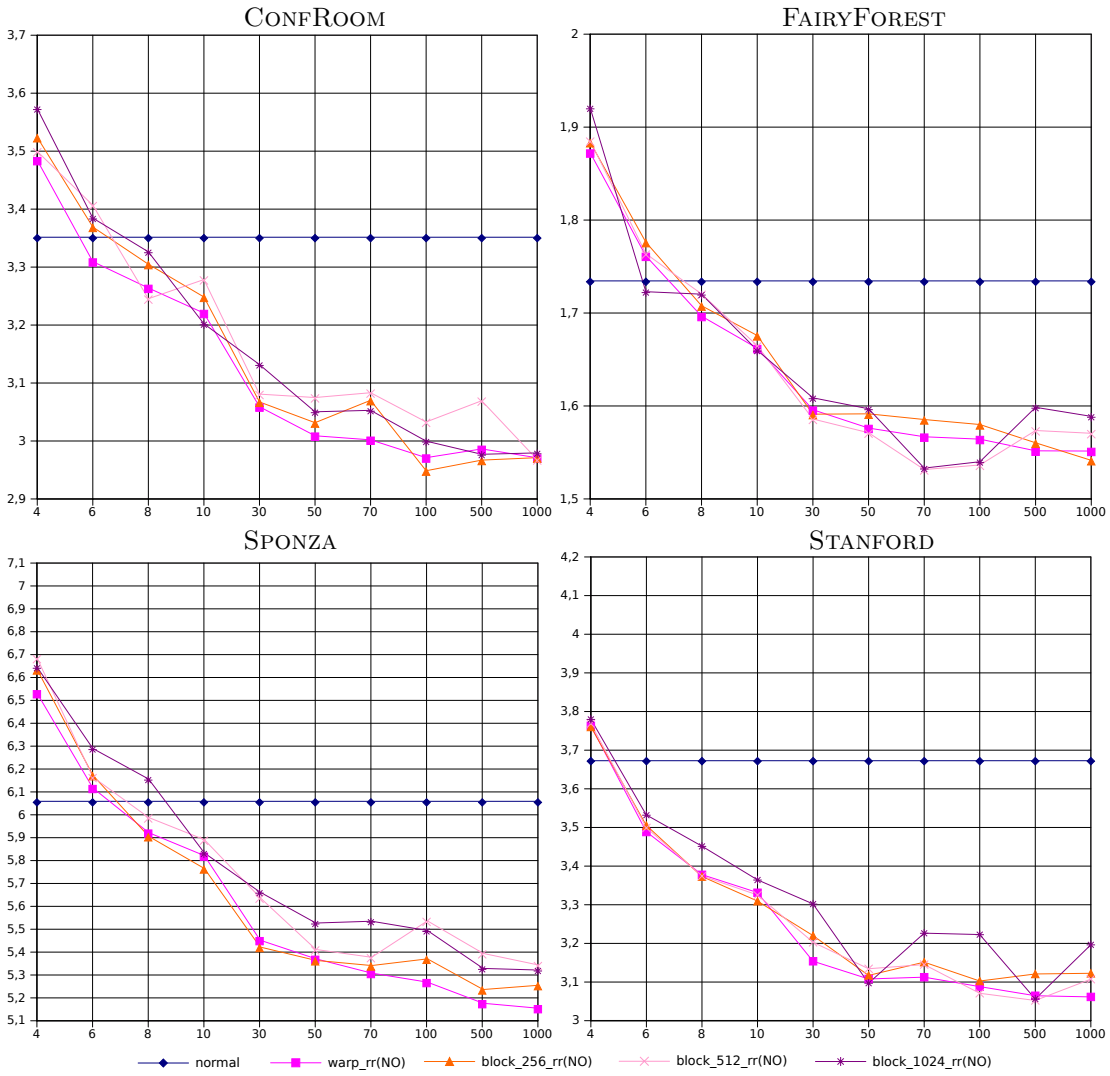


Figura 5.7: Error cuadrado medio (RMS) en % usando un intercalado de $D = 3$.

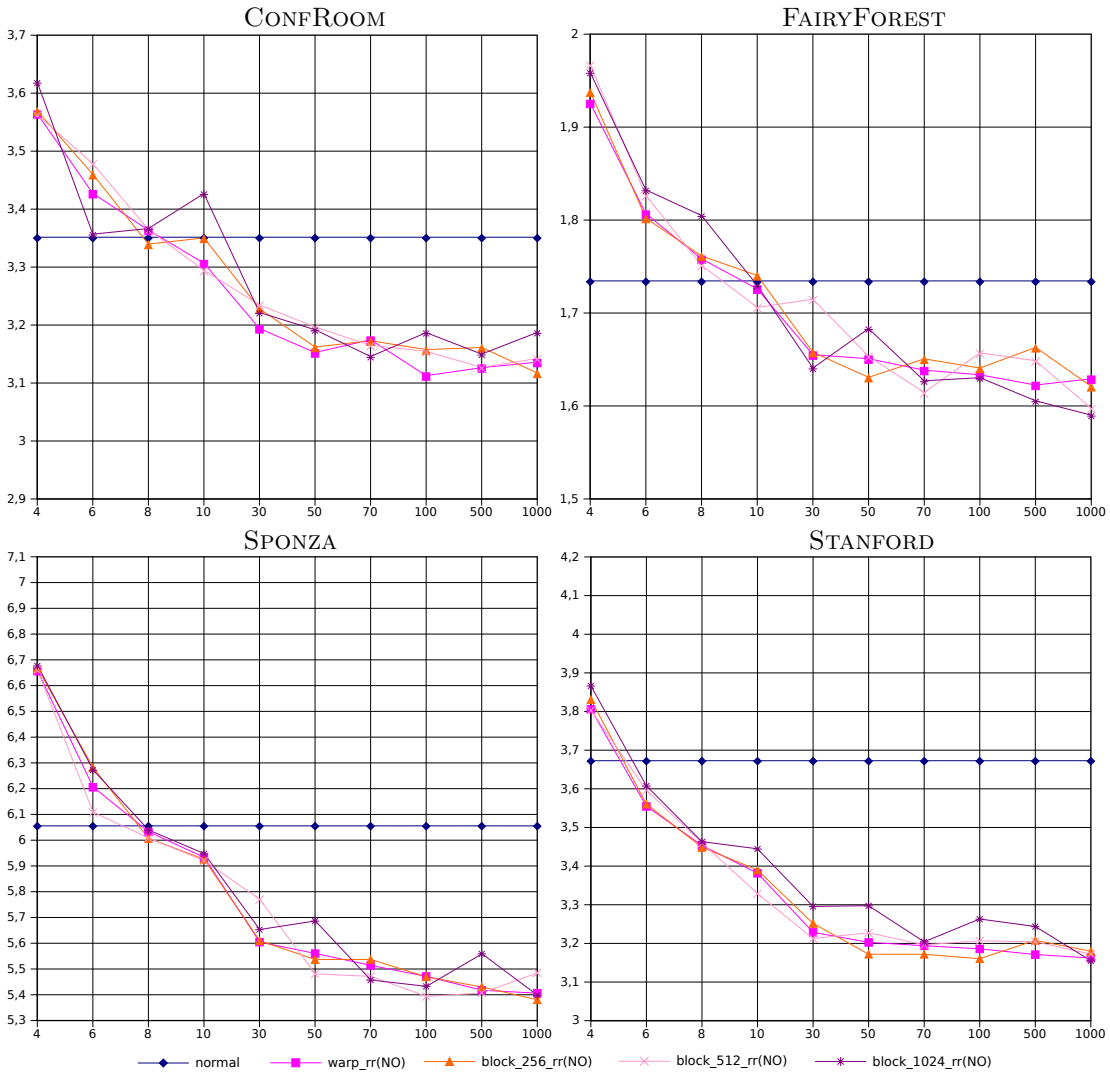


Figura 5.8: Error cuadrado medio (RMS) en % usando un intercalado de $D = 5$.

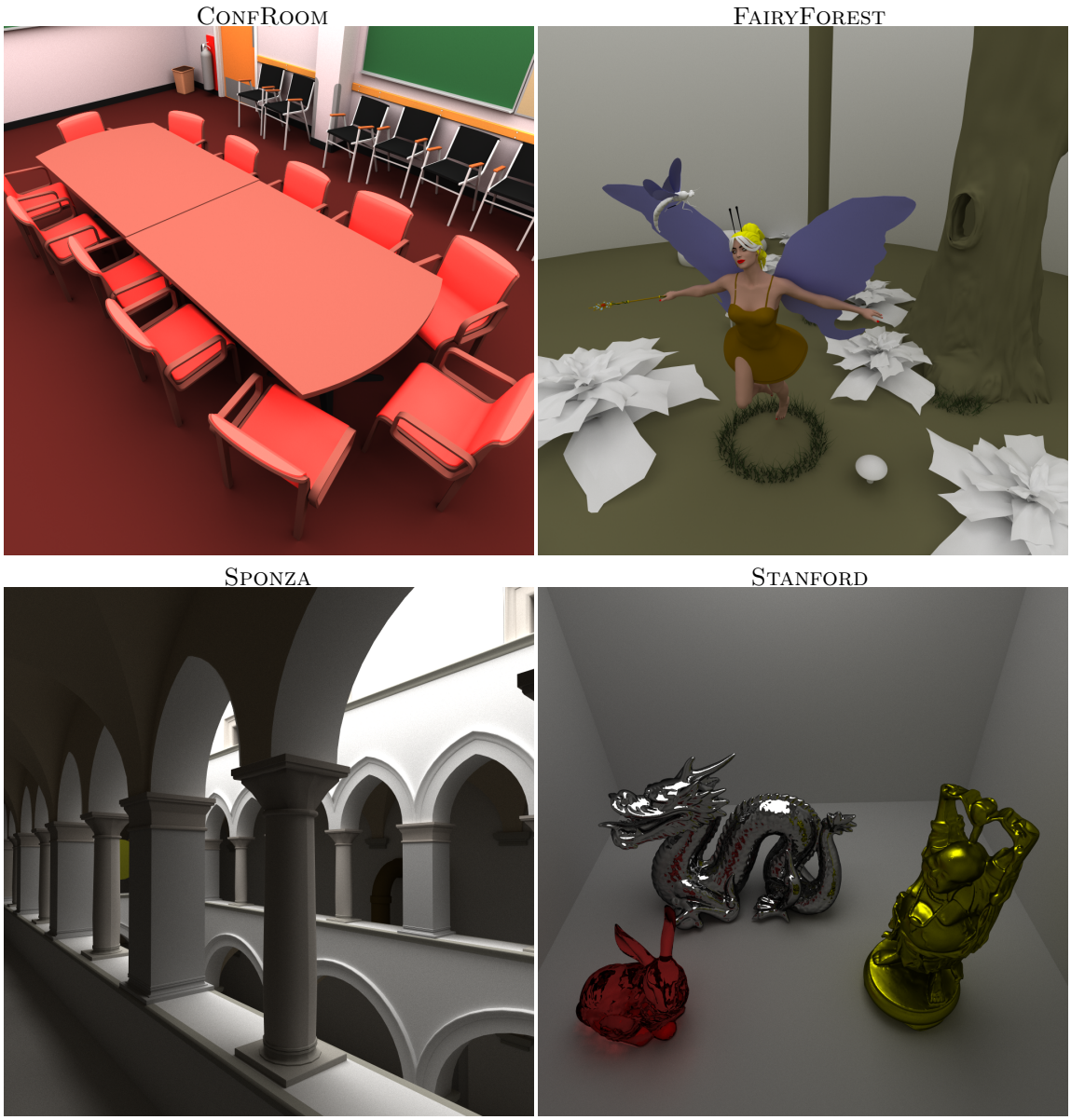


Figura 5.9: Imágenes de referencia usadas en nuestros experimentos. Han sido renderizadas con el algoritmo `normal` durante 20 minutos.

Figura 5.10: Imágenes de las escenas renderizadas con `block_1024` y ruleta rusa por grupo en 30 segundos. Se aprecia el ruido estructurado en forma de rejilla cuadrada sobre las imágenes.

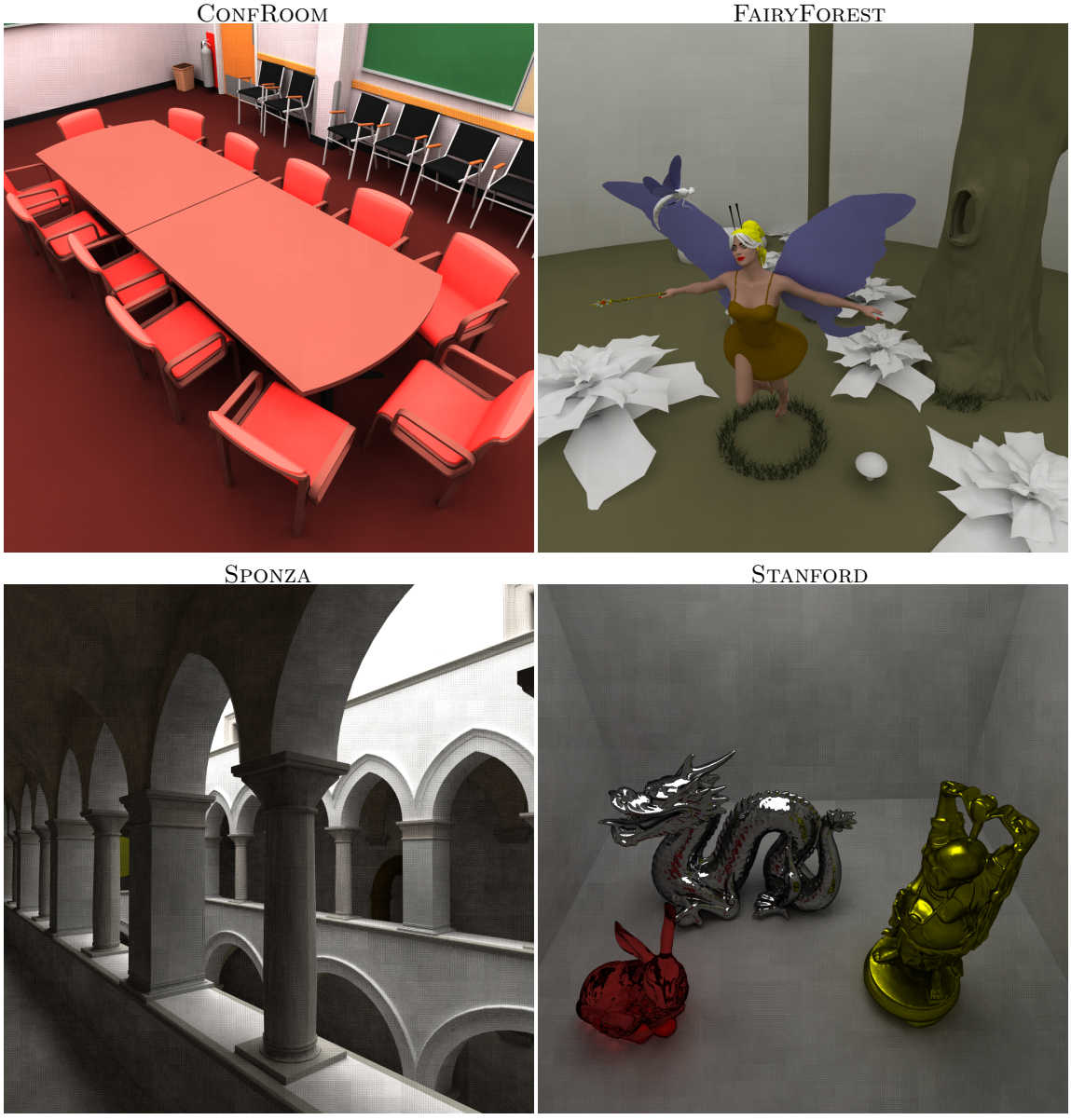


Figura 5.11: Imágenes de las escenas renderizadas con `block_1024` y ruleta rusa por grupo en 30 segundos usando un intercalado de $D = 3$. El ruido estructurado es menos apreciable que en la figura 5.10.

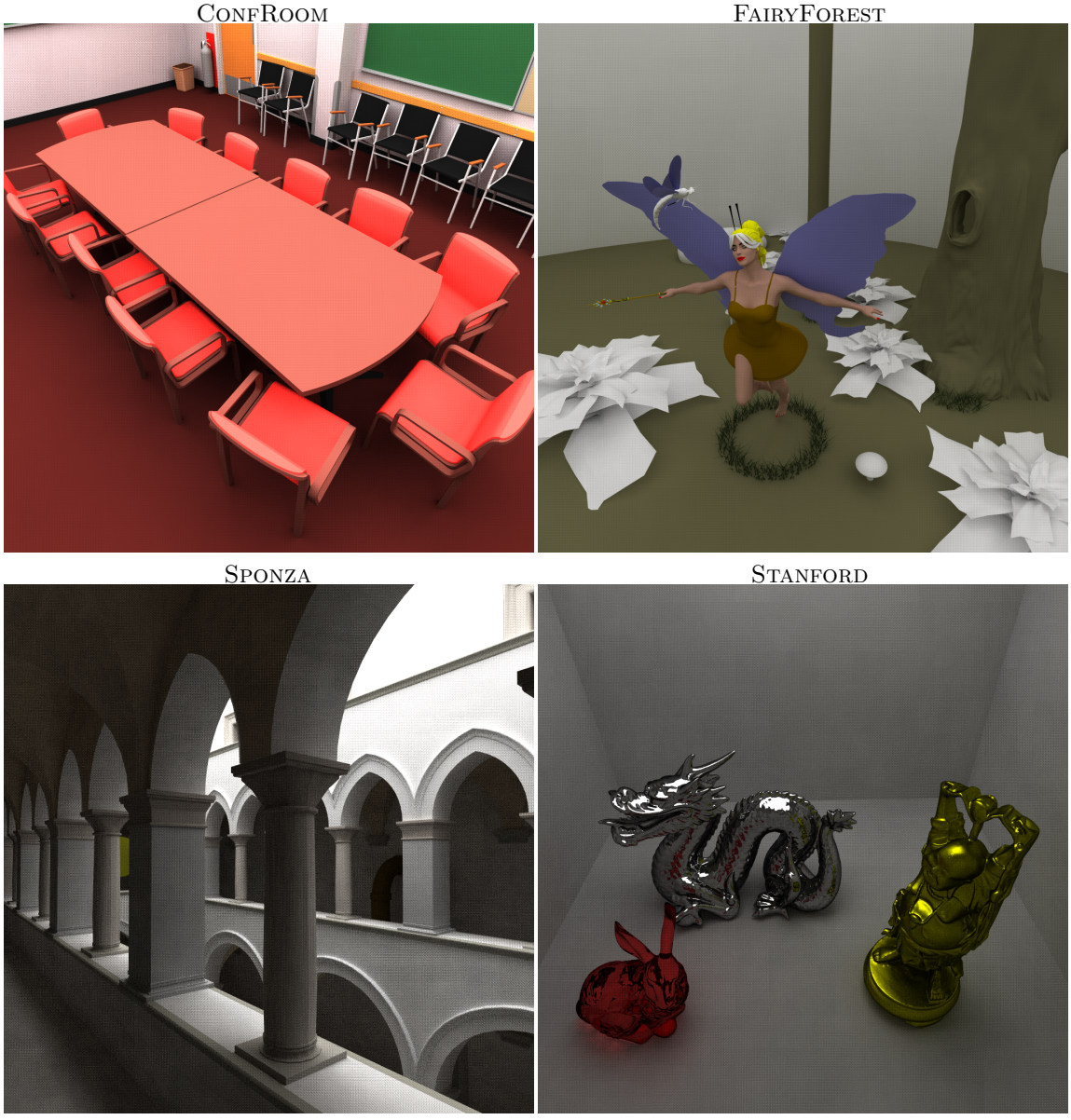


Figura 5.12: Imágenes de las escenas renderizadas con `block_1024` y ruleta rusa por grupo en 30 segundos usando un intercalado de $D = 5$. El ruido estructurado es menos apreciable que en las figuras 5.10 y 5.11.

Conclusiones y Trabajo Futuro

Los procesadores paralelos van a tener mucha importancia en los próximos años y, seguramente, mucha más a largo plazo. La ley de Moore [Moo65] se ha cumplido desde que fue citada en 1965 y a día de hoy todavía se sigue cumpliendo. Cada año, más y más componentes se pueden introducir dentro de un chip, y una parte de ellos se dedica a implementar más unidades funcionales. Esta tendencia es más notable en las GPUs en comparación con las CPUs, ya que las GPUs están orientadas a realizar cómputos en paralelo sobre una gran cantidad de datos, a diferencia de las CPUs, que dedican muchos transistores a mejorar el rendimiento de un único flujo de instrucciones [NVib]. Por ejemplo, una de las últimas GPUs en salir al mercado en el momento en que se escriben estas líneas, la GeForce 680 GTX, posee ya 1536 cores [gef] lo que le proporciona un rendimiento teórico de más de 3 teraflops [NVib].

Programación de las GPUs

Primitivas paralelas

Aunque el rendimiento teórico de las GPUs es grande, aprovechar todo este potencial de cálculo no es trivial. El programador se encuentra con la necesidad de conocer muchos aspectos técnicos del hardware para realizar una correcta codificación de su programa. Para facilitar esta tarea, se han desarrollado funciones paralelas llamadas *primitivas*. Estas funciones realizan tareas sencillas en GPU sobre arrays de datos en paralelo, y pueden servir como bloques de construcción de otros algoritmos más sofisticados. Algunas de estas primitivas se presentaron en la sección 2.4, y ya se encuentran implementadas en CUDA y disponibles en algunas librerías, tales como CUDPP [HOS+10] o Thrust [thr12].

Primitiva reducción

El rendimiento final de un algoritmo construido con estas primitivas depende, en gran medida, del rendimiento de estas funciones. En la sección 2.5, analizamos el rendimiento de tres implementaciones de la primitiva *reducción* que se pueden encontrar en la literatura. En su versión no segmentada, el algoritmo secuencial, llamado `Matrix`, es el que obtiene mejor rendimiento frente a los basados en árboles, llamados tree-based. Sin embargo, el rendimiento en la versión segmentada es altamente dependiente de los datos de entrada. Concretamente, si los segmentos son grandes, el algoritmo secuencial `Matrix` es nuevamente el que obtiene mejor rendimiento. Si los segmentos son pequeños, entonces el rendimiento de los algoritmos tree-based supera al del secuencial.

En el caso segmentado, sería deseable un algoritmo que se encargara de seleccionar el algoritmo de reducción que mejor se adaptase a la entrada en función de la distribución de sus segmentos. Desgraciadamente, este algoritmo supondría una sobrecarga añadida que podría no compensar su uso. Así, el programador decidiría si usar el algoritmo de selección automática o elegir por sí mismo el algoritmo a aplicar (secuencial o tree-based).

Por otra parte, hemos implementado y probado dos mejoras en el algoritmo secuencial de reducción: la técnica de los *bloques persistentes* y el esquema *productor-consumidor*. Con los bloques persistentes, cada bloque de hilos reduce más de un bloque de datos, lo que disminuye el número de pasadas y el almacenamiento de resultados intermedios en memoria global. Con el esquema productor-consumidor, se simultanea la reducción de un bloque con la lectura del siguiente.

Ambas mejoras algorítmicas permiten aumentar la velocidad de la reducción secuencial para esta primitiva no segmentada. Sin embargo, solo los bloques persistentes aumentan la velocidad de la reducción secuencial segmentada ya que el esquema productor-consumidor posee una ocupación mucho menor debido a su gran consumo de memoria compartida.

Primitiva scan

Todas las técnicas presentadas para implementar la primitiva reducción pueden ser igualmente empleadas para el algoritmo de *suma de prefijos* o *scan*. Esta función es la base de otras primitivas, tales como la compactación (sección 2.4.2) o la ordenación por radix (sección 2.4.4), por lo que mejorar su rendimiento supondría también un aumento del rendimiento de estas otras funciones. Queda como trabajo futuro realizar una comparación entre una implementación de scan basada en árboles, como la de Sengupta et al. [SHG08], y una secuencial, como la de Dotsenko et al. [DGS+08].

Algoritmo de Dijkstra

Para aprovechar el paralelismo de las GPUs, los algoritmos deben rediseñarse para beneficiarse de las características de este hardware. En [MTG09], hemos propuesto dos versiones paralelas del algoritmo de Dijkstra, que encuentra el camino más corto desde un nodo de un grafo a todos los demás. Estas propuestas se benefician del paralelismo de la GPU mediante el uso de fronteras compuestas, la primitiva paralela de reducción y lanzando un hilo por cada nodo.

En la primera versión, llamada `F`, cada nodo comprueba si pertenece a la frontera y, en caso afirmativo, relaja la estimación del camino más corto de sus nodos adyacentes. Esta versión requiere que el hardware disponga de operaciones atómicas para evitar el conflicto que aparece cuando dos nodos de la frontera intentan relajar el mismo nodo no resuelto. En la segunda versión, llamada `U`, los nodos no resueltos consultan todos sus nodos predecesores y comprueban si alguno de ellos pertenece a la frontera, en cuyo caso relajan su estimación. A diferencia de la versión anterior, esta no requiere operaciones atómicas.

Ambas versiones son más rápidas que la implementación en CPU del algoritmo de Dijkstra de frontera simple basada en montículos de Fibonacci. Además, el rendimiento de ambas versiones varía en función del tamaño de las fronteras compuestas que van surgiendo. Así, tanto en las versiones `U` como `F`, cuanto mayor sea el grado de los nodos, mayores tenderán a ser las fronteras y menos iteraciones serán necesarias para completar el algoritmo. Sin embargo, en las versiones `F`, una frontera grande aumenta la probabilidad de que se produzcan colisiones por los accesos atómicos a memoria, lo que penaliza su rendimiento frente a las versiones `U`.

Como trabajo futuro queda evaluar qué impacto tendría, en el rendimiento del algoritmo de Dijkstra, la introducción de la primitiva de reducción secuencial con nuestras dos mejoras algorítmicas (sección 2.5.3 y Martín et al. [MATG12]). La mejora dependerá directamente del peso de la reducción en relación al cómputo global del algoritmo.

Ray Tracing sobre GPU

Los algoritmos de ray tracing permiten obtener imágenes foto-realistas combinando, de manera sencilla, diferentes efectos visuales. El inconveniente de estos algoritmos es la necesidad de trazar

gran cantidad de rayos para obtener imágenes con suficiente calidad. Debido a que cada rayo es independiente de los demás, una implementación paralela de estos algoritmos en GPU parece sencilla. Sin embargo, esta implementación directa no aprovecha completamente el hardware.

Sistema de memoria de las GPUs

La memoria off-chip de las GPUs es más lenta que la velocidad de cómputo del chip. Dicho con otras palabras, la memoria no es capaz de proporcionar datos con suficiente velocidad para mantener activas a las unidades funcionales (Kirk y Hwu [KH10], págs. 78–79). La técnica que se usa para aliviar esta desventaja consiste en guardar los datos traídos en memorias rápidas on-chip (registros, memoria compartida o caché) y reutilizarlos en cómputos posteriores. Esta técnica aumenta la relación entre la cantidad de cómputo realizado con respecto a los datos traídos desde la memoria off-chip.

En el algoritmo de recorrido de los rayos, los datos consultados de memoria son, principalmente, los nodos de la estructura de aceleración y los triángulos de la escena. Si tenemos en cuenta el recorrido con pila de un rayo, observamos que apenas se reusa la información traída de memoria ya que los nodos se consultan solo una vez. En el recorrido con paquetes sobre una BVH hilvanada ocurre lo mismo pero a nivel de paquete, es decir, el paquete no pide a memoria dos veces el mismo nodo.

Coherencia de un lote de rayos

Sin embargo, si tenemos en cuenta el recorrido de un lote de rayos en paralelo, sí existen oportunidades de reutilización de los datos traídos desde memoria, aunque estas oportunidades dependen del comportamiento de los rayos durante el recorrido. Así, si un conjunto de rayos es coherente, va a consultar casi los mismos nodos durante el recorrido, lo que permite amortiguar el coste de las lecturas de memoria y mejorar el uso de las cachés. Si los rayos son incoherentes, no se aprovechará apenas los accesos a memoria y, consecuentemente, disminuirá el rendimiento global del sistema.

BVH hilvanada

En esta tesis hemos desarrollado tres algoritmos dedicados a aprovechar el sistema de memoria mediante la reutilización de datos. En el primero, se ha desarrollado un recorrido con paquetes sobre una BVH hilvanada (ver sección 4.5 y Torres et al. [TMG09a]). En este recorrido, cada petición de un nodo a la memoria se usa por todos los rayos activos dentro de un paquete. Este procedimiento es más eficiente cuantos más rayos del paquete se encuentren activos. Además, este recorrido también posibilita que las lecturas de memoria se adapten al estricto patrón de acceso requerido por las GPUs de cap 1.0 para que sean fusionadas.

Además, para aumentar todavía más la carga de cómputo con respecto a la misma cantidad de datos leídos de memoria, hemos propuesto la lectura de dos nodos en una misma transacción. Estos nodos forman el siguiente nodo que requiere el paquete de rayos en su recorrido y su hijo izquierdo. De esta forma, si el siguiente nodo del paquete se actualiza a su hijo izquierdo, este ya se encontrará en memoria compartida, lo que sirve para ahorrar una transacción de memoria.

Recorrido de una BVH mediante un corte

El segundo algoritmo que hemos propuesto (ver sección 4.6 y Torres et al. [TMG11]) consiste en recorrer secuencialmente una serie de subárboles (llamados *corte*) de una BVH previamente construida. Esto permite disminuir las peticiones a memoria al saltarse del recorrido los nodos cercanos a la raíz de la BVH. Además, también permite aumentar la coherencia de los rayos que

visitan cada subárbol del corte, lo que supone lecturas fusionadas y aciertos de caché. Este recorrido es adecuado, sobre todo para rayos generados tras varios rebotes en la escena.

Generación de rutas coherentes

El tercer algoritmo consiste en modificar el algoritmo de generación y terminación de rutas (ver capítulo 5 y Torres et al. [TMG12]). Esto permite generar rayos que sean geométricamente similares y preservar la coherencia de los rayos primarios tras sucesivos rebotes. De esta forma, más rutas tardan menos en recorrer la estructura de aceleración, y se obtienen imágenes con mayor calidad en el mismo margen de tiempo.

Estructuras de Aceleración

La estructura de aceleración (*EA*) es también un factor a tener en cuenta en el rendimiento de los algoritmos de *ray tracing*. Estas estructuras afectan al número medio de nodos que los rayos tienen que recorrer hasta encontrar sus puntos de intersección más cercanos. De esta manera, cuantos menos nodos y triángulos tenga que intersecar cada rayo, más rápido será el recorrido.

Construcción de estructuras de aceleración

El algoritmo *top-down* junto con la *Heurística del Área de la Superficie* (*SAH*) son la base sobre la que se construyen estructuras de aceleración jerárquicas (sección 3.4). Actualmente, las estructuras construidas con este método, o alguna de sus variaciones, son las que ofrecen mejor rendimiento en la práctica. Sin embargo, no se ha demostrado todavía que no existan otros métodos de construcción que, en un tiempo razonable, ofrezcan estructuras de aceleración con mejor rendimiento.

Heurísticas especializadas

En la sección 3.9 propusimos una variación del coste SAH, orientada a rayos cuyas direcciones se encuentran restringidas sobre un conjunto. Esto condujo a un rendimiento mayor a costa de mantener tres estructuras especializadas. Este consumo extra de memoria debe ser tenido en cuenta por el diseñador de la aplicación, que debe tomar la decisión de usar una estructura general o varias especializadas.

Compatibilidad de las heurísticas basadas en la SAH

Se han desarrollado muchas variaciones de la SAH modificando ciertas suposiciones sobre la forma de los rayos o sobre el recorrido de estos por la EA (secciones 3.8 y 3.9). Estas nuevas heurísticas han llevado a obtener mejores EAs con respecto a la SAH clásica. Es necesario examinar la compatibilidad de todos estos métodos entre sí y comprobar experimentalmente su eficacia.

Aproximación del coste teniendo en cuenta la ejecución de los rayos en la GPU

El coste SAH de una estructura de aceleración aproxima el coste medio por rayo de esa estructura. Sin embargo, no tiene en cuenta las interacciones que se producen entre los rayos durante su recorrido en GPU en cuanto a lecturas fusionadas y aciertos de caché. Introducir esta corrección en la función de coste podría producir estructuras más adecuadas a las GPUs y mejorar su rendimiento.

Conclusión Final

Las imágenes renderizadas con los algoritmos de ray tracing poseen una calidad superior a las imágenes obtenidas con el algoritmo de la tubería gráfica. Los algoritmos de ray tracing han sido tradicionalmente lentos aunque actualmente se han investigado técnicas para mejorar su rendimiento. Estas técnicas se pueden clasificar en dos grupos. El primero consiste en el desarrollo de nuevos algoritmos de recorrido que aprovechen más eficientemente el hardware sobre el que se ejecutan. El segundo se ocupa de nuevas estructuras de aceleración que permitan encontrar antes el punto de intersección más cercano de los rayos.

La aparición de las GPUs como hardware paralelo completamente programable ha hecho posible aumentar en gran medida el rendimiento de los algoritmos de ray tracing. Es de esperar que el hardware futuro posea aún más potencia de cálculo gracias al aumento del número de unidades funcionales. Los programadores sacarán el máximo provecho de esos dispositivos inventando nuevas implementaciones de los algoritmos de ray tracing, o adaptando las ya existentes. Si esta tendencia sigue a este ritmo, es posible que, en no mucho tiempo, el ray tracing sustituya a la tubería gráfica como algoritmo preferido en la generación de imágenes por computador.

Glosario

AABB Axis-aligned bounding box, o caja recubridora alineada con los ejes.

AO Ambient occlusion.

BPT Bidirectional path tracing.

BVH Bounding volume hierarchy, o jerarquía de volúmenes recubridores.

caja Volumen recubridor cuyas caras están alineadas con los ejes.

cap. Computer capability.

cdf. Función de densidad de probabilidad acumulada.

CPU Central proccess unit.

EA Estructura de aceleración.

far Nodo más lejano, en el sentido del rayo, durante el recorrido por una estructura de aceleración.

FPS Frames per second.

frustum Volumen con forma de pirámide truncada.

GPU Graphics proccessing unit.

GRC Generación de rayos coherentes.

$\mathcal{H}$ Hemisferio canónico. Hemisferio centrado en el origen de coordenadas cuyo polo se encuentra en el punto $(0, 0, 1)$.

$\mathcal{H}_x$ Hemisferio centrado en el punto x y cuya normal N_x apunta a su polo.

KD-Tree Estructura de aceleración cuyos nodos poseen un plano alineado con los ejes.

k_{pl} Dimensión del plano divisor de un nodo de un KD-Tree.

MC Monte Carlo.

MIMD Multiple instruction multiple data. Hardware en el que diferentes instrucciones se ejecutan simultáneamente sobre distintos datos.

near Nodo más cercano en el sentido del rayo durante el recorrido por una estructura de aceleración.

pdf. Función de densidad de probabilidad.

plano candidato Plano coplanario a una cara de una caja recubridora de un objeto.

plano candidato perfecto Plano candidato que proporciona el menor coste SAH.

plano divisor Plano alineado con los ejes que divide a la escena en dos partes.

p_{pl} Posición del plano divisor de un nodo en un KD-Tree.

PT Path tracing.

rayo Dado un origen $o \in \mathbb{R}^3$ y una dirección $\omega \in \mathcal{S}^2$, un rayo es el conjunto de todos los punto de la forma $o + t \cdot \omega$, donde $t > 0$.

SIMD Single instruction multiple data. Hardware en el que una misma instrucción se ejecuta simultáneamente sobre varios datos.

SM Stream multiprocessor o multiprocesador de una GPU.

$\mathcal{S}^2$ Esfera unidad centrada en el origen de coordenadas.

RRG Ruleta rusa por grupo.

RT Algoritmos de ray tracing. Comprenden todos los algoritmos que trazan rayos por una escena tridimensional para renderizar imágenes.

RTD Distributed ray tracing.

RTW Ray tracing al estilo de Whitted.

t_{end} Punto final del intervalo de un nodo durante el recorrido de un rayo por un KD-Tree.

t_{entry} Punto de entrada de un rayo en un volumen recubridor.

t_{exit} Punto de salida de un rayo en un volumen recubridor.

t_{hit} Un punto de intersección cualquiera entre un rayo y un objeto.

t_{end} Punto de comienzo del intervalo de un nodo durante el recorrido de un rayo por un KD-Tree.

t_{min} Punto de intersección más cercano entre un rayo y la escena.

t_{pl} Tiempo de intersección de un rayo con un plano divisor.

VLP Virtual Light Point.

Bibliografía

[AK87] J. Arvo y D. Kirk. Fast Ray Tracing by Ray Classification. En *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, páginas 55–64. 1987.

[AK90] J. Arvo y D. Kirk. Particle Transport and Image Synthesis. En *Proceedings of the 17th annual conference on computer graphics and interactive techniques*, SIGGRAPH'90, páginas 63–66. 1990.

[AK10] T. Aila y T. Karras. Architecture Considerations for Tracing Incoherent Rays. En *Proceedings of the High-Performance Graphics 2010*. 2010.

[AL09] T. Aila y S. Laine. Understanding the Efficiency of Ray Traversal on GPUs. En *High-Performance Graphics 2009*, páginas 145–149. 2009.

[ALK12] T. Aila, S. Laine y T. Karras. Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum. Informe técnico NVR-2012-02, NVIDIA Corporation, 2012.

[Ama84] J. Amanatides. Ray Tracing with Cones. En *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH'84, páginas 129–135. 1984.

[AMHH08] T. Akenine-Möller, E. Haines y N. Hoffman. *Real-Time Rendering*. A. K. Peters, Ltd., 3ª edición, 2008.

[Ant11] D. van Antwerpen. Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU. En *High Performance Graphics*, páginas 41–50. 2011.

[Arv88] J. Arvo. Linear-Time Voxel Walking for Octrees. *Ray Tracing News*, Marzo 1988.

[Arv95] J. R. Arvo. *Analytic Methods for Simulated Light Transport*. Tesis Doctoral, Yale University, 1995.

[AW87] J. Amanatides y A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. En *Eurographics*, páginas 3–10. 1987.

[Bat68] K. E. Batcher. Sorting Networks and Their Applications. En *Proceedings of the Spring Joint Computer Conference*, AFIPS'68 (Spring), páginas 307–314. 1968.

[BEL+07] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, I. Wald y P. Shirley. Packet-Based Whitted and Distribution Ray Tracing. En *Proceedings of Graphics Interface 2007*, páginas 177–184. 2007.

[BFH+] I. Buck, T. Foley, D. Horn, J. Sugerman, P. Hanrahan, M. Houston y K. Fatahalian. BrookGPU. `http://graphics.stanford.edu/projects/brookgpu/index.html`.

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston y P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. En *SIGGRAPH*, páginas 777–786. 2004.

[BH09] J. Bittner y V. Havran. RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. En *SCCG 2009*, páginas 61–67. Mayo 2009.

[Ble90] G. E. Blelloch. Prefix Sums and Their Applications. Informe técnico, School of Computer Science, Carnegie Mellon University, Noviembre 1990.

[BWB08] S. Boulos, I. Wald y C. Benthin. Adaptive Ray Packet Reordering. *Symposium on Interactive Ray Tracing*, páginas 131–138, 2008.

[BWS06] S. Boulos, I. Wald y P. Shirley. Geometric and Arithmetic Culling Methods for Entire Ray Packets. Informe técnico UUCS-06-010, 2006.

[caua] Caustic Professional. `https://www.caustic.com`.

[caub] Caustic Professional / Frequently Asked Questions. `https://www.caustic.com/docs_faq.php`.

[cau09] Reinventing Ray Tracing. *Dr. Dobbs's*, Julio 2009. `http://www.drdobbs.com/parallel/reinventing-ray-tracing/218500694`.

[CBZ90] S. Chatterjee, G. E. Blelloch y M. Zagha. Scan Primitives for Vector Computers. En *Proceedings of the ACM/IEEE conference on Supercomputing*, páginas 666–675. 1990.

[CHH02] N. A. Carr, J. D. Hall y J. C. Hart. The Ray Engine. En *HWWS'02: Proceedings of the ACM Conference on Graphics Hardware*, páginas 37–46. 2002.

[Chu83] K. L. Chung. *Teoría Elemental de la Probabilidad y de los Procesos Estocásticos*, páginas 114–115. Editorial Reverté, S.A., 1983.

[CK04] M. Capinski y P. E. Kopp. *Measure, Integral and Probability*. Springer Undergraduate Mathematics Series. Springer, 2ª edición, 2004.

[CPC84] R. L. Cook, T. Porter y L. Carpenter. Distributed Ray Tracing. En *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH'84, páginas 137–145. 1984.

[CSN04] F. Castro, M. Sbert y L. Neumann. Fast Multipath Radiosity using Hierarchical Subscenes. *Computer Graphics Forum*, 23(1):43–54, Marzo 2004.

[CTE05] D. Cline, J. Talbot y P. Egbert. Energy Redistribution Path Tracing. En *SIGGRAPH 2005*, páginas 1186–1195. 2005.

[DGS+08] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd y J. Manferdelli. Fast Scan Algorithms on Graphics Processors. En *Proceedings of the 22nd annual international conference on Supercomputing*, ICS'08, páginas 205–213. ACM, 2008.

[DHK08] H. Dammertz, J. Hanika y A. Keller. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. En *Proceedings of the Nineteenth Eurographics conference on Rendering*, EGSR'08, páginas 1225–1233. 2008.

[DK08] H. Dammertz y A. Keller. The Edge Volume Heuristic - Robust Triangle Subdivision for Improved BVH Performance. En *Interactive Ray Tracing*, páginas 155 –158. 2008.

[DLW93] P. Dutré, E. P. Lafortune y Y. Willems. Monte Carlo Light Tracing with Direct Computation of Pixel Intensities. En *3rd International Conference on Computational Graphics and Visualisation Techniques*, páginas 128–137. 1993.

[DS12] W. L. Dunn y J. K. Shultis. *Exploring Monte Carlo Methods*. Elsevier, 1ª edición, 2012.

[DWBS03] A. Dietrich, I. Wald, C. Benthin y P. Slusallek. The OpenRT Application Programming Interface - Towards a Common API for Interactive Ray Tracing. En *Proceedings of the 2003 OpenSG Symposium*. 2003.

[EG07] M. Ernst y G. Greiner. Early Split Clipping for Bounding Volume Hierarchies. En *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, páginas 73–78. 2007.

[EG08] M. Ernst y G. Greiner. Multi Bounding Volume Hierarchies. En *IEEE Symposium on Interactive Ray Tracing*, páginas 35–40. 2008.

[Eve01] C. Everitt. Interactive Order-Independent Transparency. Informe técnico, NVIDIA, 2001.

[Fey89] R. P. Feynman. *Electrodinámica Cuántica: La Extraña Teoría de la Luz y la Materia*. Alianza Universidad. Alianza Editorial, Junio 1989.

[FFD09] B. Fabianowski, C. Flower y J. Dingliana. A Cost Metric for Scene-Interior Ray Origins. En *Eurographics 2009 Short Papers*, páginas 49–52. 2009.

[FS05] T. Foley y J. Sugerman. KD-Tree Acceleration Structures for a GPU Raytracer. En *Graphics Hardware 2005*, páginas 15–22. 2005.

[gef] GeForce GTX 680 Specifications. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/specifications`.

[GL10] K. Garanzha y C. T. Loop. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum*, 29(2):289–298, 2010.

[GPSS07] J. Günther, S. Popov, H.-P. Seidel y P. Slusallek. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. En *Interactive Ray Tracing*, páginas 113–118. 2007.

[GR08] C. P. Gribble y K. Ramani. Coherent Ray Tracing via Stream Filtering. En *IEEE/Eurographics Symposium on Interactive Ray Tracing*, páginas 59–66. 2008.

[GS87] J. Goldsmith y J. Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Application*, 7(5):14–20, 1987.

[GTGB84] C. M. Goral, K. E. Torrance, D. P. Greenberg y B. Battaile. Modeling the Interaction of Light Between Diffuse Surfaces. En *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH'84, páginas 213–222. 1984.

[Hac04] T. Hachisuka. Final Gathering on GPU. En *SIGGRAPH poster*. 2004.

[Hac05] T. Hachisuka. High-Quality Global Illumination Rendering Using Rasterization. En *GPU Gems 2*, páginas 615–633. Addison-Wesley, 2005.

[Har07a] M. Harris. Optimizing Parallel Reduction in CUDA. Informe técnico, Nvidia, 2007. `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf`.

[Har07b] M. Harris. Parallel Prefix Sum (Scan) with CUDA. Informe técnico, NVIDIA, Abril 2007.

[Hav00] V. Havran. *Heuristic Ray Shooting Algorithms*. Tesis Doctoral, Faculty of Electrical Engineering, Czech Technical University in Prague, 2000.

[HB99] V. Havran y J. Bittner. Rectilinear Trees for Preferred Ray Sets. En *SCCG*, páginas 171–178. 1999.

[HB02] V. Havran y J. Bittner. On Improving KD-Trees for Ray Shooting. *Journal of WSCG*, 10(1):209–216, Febrero 2002.

[HB07] V. Havran y J. Bittner. Ray Tracing with Sparse Boxes. En *Proceedgings of Spring Conference on Computer Graphics*, SCCG 2007, páginas 49–54. 2007.

[HBv98] V. Havran, J. Bittner y J. Žára. Ray Tracing with Rope Trees. En *Proceedings of SCCG'98 (Spring Conference on Computer Graphics)*, páginas 130–139. Budmerice, Slovak Republic, 1998.

[HDW+11] M. Hapala, T. Davidovic, I. Wald, V. Havran y P. Slusallek. Efficient Stack-less BVH Traversal for Ray Tracing. En *Proceedings 27th Spring Conference of Computer Graphics (SCCG) 2011*, páginas 29–34. 2011.

[HH84] P. S. Heckbert y P. Hanrahan. Beam Tracing Polygonal Objects. En *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, páginas 119–127. 1984.

[HHGM10] J. Hermes, N. Henrich, T. Grosch y S. Mueller. Global Illumination Using Parallel Global Ray Bundles. En *Vision, Modeling and Visualization*, páginas 65–72. 2010.

[HHS06] V. Havran, R. Herzog y H.-P. Seidel. On Fast Construction of Spatial Hierarchies for Ray Tracing. Informe técnico MPI-I-2006-4-004, Max-Planck-Institut für Informatik, Junio 2006.

[HKBv97] V. Havran, T. Kopal, J. Bittner y J. Žára. Fast Robust BSP Tree Traversal Algorithm for Ray Tracing. *Journal of Graphics Tools*, 2(4):15–23, 1997.

[HM08a] W. Hunt y W. R. Mark. Adaptive Acceleration Structures in Perspective Space. En *IEEE Symposium on Interactive Ray Tracing*, páginas 11–17. 2008.

[HM08b] W. Hunt y W. R. Mark. Ray-Specialized Acceleration Structures for Ray Tracing. En *IEEE/EG Symposium on Interactive Ray Tracing*, páginas 3–10. 2008.

[HN07] P. Harish y P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. En *Proceedings of the 14th international conference on High performance computing*, HiPC'07, páginas 197–208. 2007.

[Hor05] D. Horn. Stream Reduction Operations for GPGPU Applications. En *GPU Gems 2*, capítulo 36. Addison-Wesley Professional, 2005.

[HOS+08] M. Harris, J. D. Owens, S. Sengupta, S. Tseng, Y. Zhang, A. Davidson y N. Satish. CUDA Data Parallel Primitives Library (CUDPP 1.0), Abril 2008. `http://www.gpgpu.org/static/developer/cudpp/rel/cudpp_1.1/html/index.html`.

[HOS+10] M. Harris, J. D. Owens, S. Sengupta, S. Tseng, Y. Zhang, A. Davidson y N. Satish. CUDA Data Parallel Primitives Library (CUDPP 1.1.1), Abril 2010. `http://code.google.com/p/cudpp/`.

[HPJ12] T. Hachisuka, J. Pantaleoni y H. W. Jensen. A Path Space Extension for Robust Light Transport Simulation. Informe técnico NVR-2012-001, NVidia, Abril 2012.

[HS86] W. D. Hillis y G. L. Steele, Jr. Data Parallel Algorithms. *Commun. ACM*, 29(12):1170–1183, Diciembre 1986.

[HSMH07] D. R. Horn, J. Sugerman, H. Mike y P. Hanrahan. Interactive KD-Tree GPU Raytracing. En *I3D 2007*, páginas 167–174. 2007.

[HSO07] M. Harris, S. Sengupta y J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. En H. Nguyen, editor, *GPU Gems 3*, capítulo 39, páginas 851–876. Addison Wesley, 2007.

[Hun08] W. Hunt. Corrections to the Surface Area Metric with Respect to Mail-Boxing. En *Eurographics Symposium on Interactive Ray Tracing 2008*, páginas 77–80. 2008.

[IWP08] T. Ize, I. Wald y S. G. Parker. Ray Tracing with the BSP Tree. En *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*. 2008.

[Jen96] H. W. Jensen. Global Illumination Using Photon Maps. En *Proceedings of the Eurographics Workshop on Rendering Techniques*, páginas 21–30. 1996.

[Kaj86] J. T. Kajiya. The Rendering Equation. *SIGGRAPH Computer Graphics*, 20(4):143–150, 1986.

[KAL12] T. Karras, T. Aila y S. Laine. Open Source Implementation of Understanding the Efficiency of Ray Traversal on GPUs, release 1.4, Junio 2012. `http://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/`.

[Kel97] A. Keller. Instant Radiosity. En *Computer Graphics and Interactive Techniques*, páginas 49–56. 1997.

[KH01] A. Keller y W. Heidrich. Interleaved Sampling. En *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, páginas 269–276. 2001.

[KH10] D. B. Kirk y W. W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

[KK86] T. L. Kay y J. T. Kajiya. Ray Tracing Complex Scenes. En *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH, páginas 269–278. 1986.

[Lai10] S. Laine. Restart Trail for Stackless BVH Traversal. En *Proceedings of the Conference on High Performance Graphics*, HPG'10, páginas 107–111. 2010.

[LE10] H. Ludvigsen y A. C. Elster. Real-Time Ray Tracing Using Nvidia OptiX. páginas 65–68. 2010.

[LW93] E. P. Lafortune y Y. D. Willems. Bi-Directional Path Tracing. En *Proceedings of the third international conference on computational graphics and visualization techniques*, COMPUGRAPHICS'93, páginas 145–153. 1993.

[MATG12] P. J. Martín, L. F. Ayuso, R. Torres y A. Gavilanes. Algorithmic Strategies for Optimizing the Parallel Reduction Primitive in CUDA. En *2012 International Conference on High Performance Computing & Simulation*, HPCS 2012, páginas 511–519. Julio 2012.

[MB90] D. J. MacDonald y K. S. Booth. Heuristics for Ray Tracing Using Space Subdivision. *Visual Computer*, 6(3):153–166, 1990.

[MBK+10] B. Moon, Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S. W. Nam y S.-E. Yoon. Cache-Oblivious Ray Reordering. *ACM Trans. Graph.*, 29(3):1–10, 2010.

[Met87] N. Metropolis. The Beginning of the Monte Carlo Method. *Los Alamos Science*, páginas 125–130, 1987.

[MFM09] R. Martínez, A. Forés y I. Martín. Parallel Implementation of a Global Line Monte Carlo Radiosity. En *GRAPP*, páginas 164–169. 2009.

[MMAM07] E. Mansson, J. Munkberg y T. Akenine-Moller. Deep Coherent Ray Tracing. En *Symposium on Interactive Ray Tracing*, páginas 79–85. 2007.

[Moo65] G. E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), Abril 1965.

[Mor11] B. Mora. Naive Ray-Tracing: A Divide-and-Conquer Approach. *ACM Trans. Graph.*, 30(5):117:1–117:12, 2011.

[MT97] T. Möller y B. Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[MTG08] P. J. Martín, R. Torres y A. Gavilanes. Soluciones CUDA al Problema del Camino más Corto desde un Origen. En *Workshop de Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones (ANACAP)*. 2008.

[MTG09] P. J. Martín, R. Torres y A. Gavilanes. CUDA Solutions for the SSSP Problem. En *Proceedings of the 9th International Conference on Computational Science*, ICCS'09, páginas 904–913. 2009.

[NFLM07] P. A. Návratil, D. S. Fussell, C. Lin y W. R. Mark. Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization. En *Symposium on Interactive Ray Tracing*, páginas 95–104. 2007.

[NHD10] J. Novák, V. Havran y C. Daschbacher. Path Regeneration for Interactive Path Tracing. En *Eurographics, short papers*, páginas 61–64. 2010.

[Nic63] F. E. Nicodemus. Radiance. *American Journal of Physics*, 31:368–377, 1963.

[Nic65] F. E. Nicodemus. Directional Reflectance and Emissivity of an Opaque Surface. *Appl. Opt.*, 4(7):767–773, Julio 1965.

[NVia] NVidia Corporation. *CUDA C Programming Guide - Computer Capabilities*, 4.2 edición. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities`.

[NVib] NVidia Corporation. *CUDA C Programming Guide - Introduction*, 4.2 edición. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction`.

[Nvi09] Nvidia. Nvidia's Next Generation CUDA Computer Architecture: Fermi. Informe técnico, NVidia Corporation, 2009. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[NVi11] NVidia. NVIDIA CUDA C Programming Guide. Informe técnico, NVidia, Mayo 2011.

[ORM08] R. Overbeck, R. Ramamoorthi y W. R. Mark. Large Ray Packets for Real-Time Whitted Ray Tracing. En *IEEE Symposium on Interactive Ray Tracing*, páginas 41–48. 2008.

[PBD+10] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison y M. Stich. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics*, 2010.

[PBMH02] T. J. Purcell, I. Buck, W. R. Mark y P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002.

[PBPP11] A. Pajot, L. Barthe, M. Paulin y P. Poulin. Combinatorial Bidirectional Path-Tracing for Efficient Hybrid CPU/GPU Rendering. *Computer Graphics Forum*, 30(2):315–324, Abril 2011.

[PDC+05] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen y P. Hanrahan. Photon Mapping on Programmable Graphics Hardware. En *SIGGRAPH*, páginas 41–50. 2005.

[PGDS09] S. Popov, I. Georgiev, R. Dimov y P. Slusallek. Object Partitioning Considered Harmful: Space Subdivision for BVHs. En *HPG'09: Proceedings of the 1st ACM conference on High Performance Graphics*, páginas 15–22. 2009.

[PGSS06] S. Popov, J. Günther, H.-P. Seidel y P. Slusallek. Experiences with Streaming Construction of SAH KD-Trees. En *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, páginas 89–94. Septiembre 2006.

[PGSS07] S. Popov, J. Günther, H.-P. Seidel y P. Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26(3):415–424, 2007.

[PH10] M. Pharr y G. Humphreys. *Physically Based Rendering: From Theory to Implementation (second edition)*. Morgan Kaufmann, Julio 2010.

[PKGH97] M. Pharr, C. Kolb, R. Gershbein y P. Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. En *SIGGRAPH*, páginas 101–108. 1997.

[PM88] S. K. Park y K. W. Miller. Random Number Generator: Good Ones Are Hard to Find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[PMS+99] S. Parker, W. Martin, P. P. Sloan, P. Shirley, B. Smits y C. Hansen. Interactive Ray Tracing. En *ACM Symposium on Interactive 3D Graphics*, páginas 119–126. Abril 1999.

[PSL+98] S. Parker, P. Shirley, Y. Livnat, C. Hansen y P.-P. Sloan. Interactive Ray Tracing for Isosurface Rendering. En *Proceedings of the conference on Visualization*, VIS'98, páginas 233–238. 1998.

[PSS98] S. Parker, P. Shirley y B. Smits. Single Sample Soft Shadows. Informe técnico, Octubre 1998.

[Pur04] T. J. Purcell. *Ray Tracing on a Stream Processor*. Tesis Doctoral, Stanford University, CA, USA, 2004.

[RAH07] D. Roger, U. Assarsson y N. Holzschuch. Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU. En *Rendering Techniques (Proceedings of the Eurographics Symposium on Rendering)*, páginas 99–110. Junio 2007.

[ray] Siliconarts / raycore® series 1000. `http://www.siliconarts.co.kr/gpu-ip`.

[Res06] A. Reshetov. Omnidirectional Ray Tracing Traversal Algorithm for KD-Trees. En *Interactive Ray Tracing*, páginas 57–60. Septiembre 2006.

[RGD09] K. Ramani, C. P. Gribble y A. Davis. StreamRay: A Stream Filtering Architecture for Coherent Ray Tracing. En *Internationa Conference on Architectural Support for Programming Languajes and Operating System*, páginas 325–336. 2009.

[RRB+08] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk y W. mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. En S. Chatterjee y M. L. Scott, editores, *PPoPP*, páginas 73–82. ACM, 2008.

[RSH05] A. Reshetov, A. Soupikov y J. Hurley. Multi-Level Ray Tracing Algorithm. En *SIGGRAPH*, páginas 1176–1185. 2005.

[RW80] S. M. Rubin y T. Whitted. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. En *SIGGRAPH*, páginas 110–116. 1980.

[SCJ09] I. Sadeghi, B. Chen y H. W. Jensen. Coherent Path Tracing. *J. Graphics, GPU, & Game Tools*, 14(2):33–43, 2009.

[SCS+08] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan y P. Hanrahan. Larrabee: a Many-core x86 Architecture for Visual Computing. En *SIGGRAPH 2008*, volumen 18, páginas 1–15. 2008.

[SFD09] M. Stich, H. Friedrich y A. Dietrich. Spatial Splits in Bounding Volume Hierarchies. En *Proceedings of the Conference on High Performance Graphics 2009*, HPG'09, páginas 7–13. 2009.

[She10] M. C. Shebanow. The Fermi Architecture. Informe técnico, NV Research, 2010.

[SHG08] S. Sengupta, M. Harris y M. Garland. Efficient Parallel Scan Algorithms for GPUs. Informe técnico, NVIDIA, 2008.

[SHG09] N. Satish, M. Harris y M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. En *IPDPS*, páginas 1–10. IEEE, 2009.

[SHGO11] S. Sengupta, M. Harris, M. Garland y J. D. Owens. Efficient Parallel Scan Algorithms for Many-core GPUs. En *Scientific Computing with Multicore and Accelerators*, capítulo 19, páginas 413–442. Enero 2011.

[Shi91] P. S. Shirley. *Phisically Based Lighting Calculations for Computer Graphics*. Tesis Doctoral, University of Illinois, 1991.

[Shi94] P. Shirley. Hybrid Radiosity/Monte Carlo Methods. En *SIGGRAPH'94 Course Notes*. 1994.

[SHZO07] S. Sengupta, M. Harris, Y. Zhang y J. D. Owens. Scan Primitives for GPU Computing. En *Graphics Hardware*, páginas 97–106. 2007.

[sil] Siliconarts. `http://www.siliconarts.com/`.

[SIP07] B. Segovia, J.-C. Iehl y B. Péroche. Coherent Metropolis Light Transport with Multiple-Try Mutations. Informe técnico, Abril 2007.

[SKP98] L. Szirmay-Kalos y W. Purgathofer. Global Ray-bundle Tracing with Hardware Acceleration. En *Rendering Techniques*, páginas 247–258. 1998.

[SLO06] S. Sengupta, A. Lefohn y J. D. Owens. A Work-Efficient Step-Efficient Prefix Sum Algorithm. En *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, páginas 26–27. Mayo 2006.

[SLS03] J. Schmittler, A. Leidinger y P. Slusallek. A Virtual Memory Architecture For Real-Time Ray Tracing Hardware. En *Computer and Graphics 27, 5*, páginas 693–699. Octubre 2003.

[SM03] P. Shirley y R. K. Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., 2003.

[Smi98] B. Smits. Efficiency Issues for Ray Tracing. *Journal of Graphics Tools*, 3:1–14, 1998.

[SMP98] M. Sbert, R. Martínez y X. Pueyo. Gathering Multi-Path: a New Monte Carlo Algorithm for Radiosity. En *WSCG'98*. Febrero 1998.

[SWS02] J. Schmittler, I. Wald y P. Slusallek. SaarCOR – A Hardware Architecture for Real-time Ray-Tracing. En *Proceedings of Eurographics Workshop on Graphics Hardware*. 2002.

[SWW+04] J. Schmittler, S. Woop, D. Wagner, W. J. Paul y P. Slusallek. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. En *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS'04, páginas 95–106. 2004.

[thr12] Thrust: Code at the Speed of Light. Release 1.6.0, Marzo 2012. `http://code.google.com/p/thrust/`.

[TMG08] R. Torres, P. J. Martín y A. Gavilanes. Ray Tracing en CUDA usando una BVH Hilvanada. En *Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones (ANACAP)*. 2008.

[TMG09a] R. Torres, P. J. Martín y A. Gavilanes. Ray Casting using a Roped BVH with CUDA. En *Proceedings of the Spring Conference on Computer Graphics*, SCCG, páginas 107–114. 2009.

[TMG09b] R. Torres, P. J. Martín y A. Gavilanes. Ray Tracing en CUDA usando una BVH Múltiple. En *Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones (ANACAP)*. 2009.

[TMG11] R. Torres, P. Martin y A. Gavilanes. Traversing a BVH Cut to Exploit Ray Coherence. En *GRAPP 2011*, páginas 140–150. 2011.

[TMG12] R. Torres, P. J. Martín y A. Gavilanes. Generating Coherent Ray Directions in Path Tracing. En *Ceig 2012, XXII Spanish Computer Graphics Conference*, páginas 80–90. 2012.

[TMGA12] R. Torres, P. J. Martín, A. Gavilanes y L. F. Ayuso. Improving Ray Traversal by Using Several Specialized KD-Trees. En *GRAPP/IVAPP*, páginas 215–226. 2012.

[TO12] Y. Tokuyoshi y S. Ogaki. Real-time Bidirectional Path Tracing via Rasterization. En *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D'12, páginas 183–190. 2012.

[Tsa09] J. A. Tsakok. Faster Incoherent Rays: Multi-BVH Ray Stream Tracing. En *Proceedings of the Conference on High Performance Graphics 2009*, HPG'09, páginas 151–158. 2009.

[Vea98] E. Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. Tesis Doctoral, Stanford University, 1998.

[VG94] E. Veach y L. Guibas. Bidirectional Estimators for Light Transport. En *Eurographics Rendering Workshop*. 1994.

[VG97] E. Veach y L. J. Guibas. Metropolis Light Transport. En *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, páginas 65–76. 1997.

[Wal04] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. Tesis Doctoral, Saarland University, Germany, 2004.

[Wal07] I. Wald. On Fast Construction of SAH-Based Bounding Volume Hierarchies. En *Symposium on Interactive Ray Tracing 2007*, páginas 33–40. 2007.

[WBB08] I. Wald, C. Benthin y S. Boulos. Getting Rid of Packets - Efficient SIMD Single-Ray Traversal Using Multi-Branching BVHs. En *Interactive Ray Tracing*, páginas 49–57. 2008.

[WBMS05] A. Williams, S. Barrus, R. K. Morley y P. Shirley. An Efficient and Robust Ray-Box Intersection Algorithm. *Journal of Graphics Tools*, 10(1):49–54, Junio 2005.

[WBS07] I. Wald, S. Boulos y P. Shirley. Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[WBWS01] I. Wald, C. Benthin, M. Wagner y P. Slusallek. Interactive Rendering with Coherent Ray Tracing. En *Computer Graphics Forum (Proceedings of Eurographics'01)*, volumen 20, páginas 153–164. 2001.

[WGBK07] I. Wald, C. P. Gribble, S. Boulos y A. Kensler. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Informe técnico, 2007.

[WGS04] I. Wald, J. Günther y P. Slusallek. Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristic. *Computer Graphics Forum*, 22(3):595–603, 2004.

[WH06] I. Wald y V. Havran. On Building Fast KD-Trees for Ray Tracing, and on Doing That in $O(N \log N)$. En *Symposium on Interactive Ray Tracing*, páginas 61–69. 2006.

[Whi80] T. Whitted. An Improved Illumination Model for Shaded Display. *Commun. ACM*, 23(6):343–349, 1980.

[WKB+02] I. Wald, T. Kollig, C. Benthin, A. Keller y P. Slusallek. Interactive Global Illumination Using Fast Ray Tracing. En *Eurographics Workshop on Rendering*, páginas 15–24. 2002.

[Woo04] S. Woop. *A Ray Tracing Hardware Architecture for Dynamic Scenes*. Tesis de Master, Universität des Saarlandes, Saarbrücken, Germany, 2004.

[WSS05] S. Woop, J. Schmittler y P. Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. En *SIGGRAPH*, páginas 434–444. 2005.

[ZB91] M. Zagha y G. E. Blelloch. Radix Sort for Vector Multiprocessors. En *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing'91, páginas 712–721. 1991.

[ZHWG08] K. Zhou, Q. Hou, R. Wang y B. Guo. Real-time KD-tree Construction on Graphics Hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.

[ZK99] A. Zomaya y R. Kazman. *Handbook of Algorithms and Theory of Computation*, capítulo Simulated Annealing Techniques, páginas 37.1–37.19. CRC Press, 1999.

Parte II

Extensive English Summary

Chapter 6

Computational Model of Light and Matter

6.1. Introduction

One of the aims of *Computer Graphics* is that a computer is able to render images subjectively indistinguishable from a photography. First of all, in order to obtain this kind of images, it is necessary to define a computational model describing the behaviour of light in the real world. So, we have to determine how our three-dimensional *objects* are defined in our *scene* and how light interacts with them.

6.2. Particle Model of Light

In order to generate realistic images, it is not necessary to simulate the behaviour of light according to the present physics model. An easier model, based on *radiant energy*, is enough to obtain a high level of realism in the computer-generated images. Unfortunately, this model excludes some light phenomenons such as *polarization*, *interference* or *phosphorescence*. However, these phenomenons are not very significant and they can be simulated other way within this model.

Suppose light is composed of punctual particles that travel along a straight line at the same velocity and they do not interact each other. Each of them is defined by a point in the space, a velocity vector and a wavelength. These particles are generated on the surfaces of the objects and they travel along a straight line to other surfaces. At the hit time, each particle is either absorbed by the object (then it disappears) or thrown until it reaches another surface again. The latter event is called *scattering*. The new direction of this particle is randomly chosen within a range of directions, which only depends on the material of the object. During the lifetime of the particles, some of them are possible to reach the camera, where they are detected by one or several pixels of the film and contribute to the final image.

6.3. Particle Measurement

In order to derive the magnitudes we will use during the rendering, we have to measure the among of particles that can be found in a certain volume at a certain time lapse. We assume that the number of particles is so high that we can employ calculus to measure them. Specifically, we will derive *radiance* (or radiant energy) as the fundamental magnitude.

A set of moving particles defines a volume in the space in a time lapse. Think about the set of particles that passes though a surface S from inside to outside. Moreover, that set of particles is shrunk to consider only those particles whose directions are in the set of directions Ω. In a certain time lapse $[t_0, t_1]$, the previous set of particles forms a cone-shaped volume. To measure that volume, we are going to utilize usual measure functions on surfaces, directions and time. We use *area* as measure on surface, denoted as A and measured in squared meters $[m^2]$. We use the length t of the time lapse, measured in seconds $[s]$. The directions are points on the surface of the unit sphere, so they are measured with the solid angle, denoted as σ and measured in steradians $[sr]$.

We define the measure function η as the "amount" of particles that passes through the surface S whose directions are in Ω and in the time lapse $[t_0, t_1]$ as follows

$$\eta(S \to \Omega, [t_0, t_1]) = \int_{t_0}^{t_1} \int_{\omega \in \Omega} \int_{x \in S} 1 \, dA(x) \, d\sigma(\omega) \, dt$$

The arrow $\to$ indicates that the particles pass through the surface from inside to outside. Symmetrically, we use the arrow $\leftarrow$ if the particles arrive on the surface.

6.3.1. Derivation of the Radiance

The most common measure for a set of particles is their *energy*, denoted as Q and measured in *Joules* $[J]$. We assume that Q and η are *absolutely continuous* (denoted as $Q \ll \eta$), i.e. any set of particles being null-set ($\eta = 0$) does not carry energy ($Q = 0$). This seems reasonable because if a volume is empty, it does not contain any particle, so it does not carry energy.

Due to the fact that these measures are absolutely continuous, there exists a density function that connects Q and η. This density function is called *radiance* and it is usually denoted as L (Nicodemus [Nic63]). The function L connects the measures Q and η as

$$Q = \int L \, d\eta$$

It is usual to use the derivative notation to define L

$$L(x \to \omega, t) = \frac{dQ}{d\eta} = \frac{dQ}{dA^{\omega}(x) \, d\sigma_x(\omega) \, dt} \tag{6.1}$$

where $dA^{\omega}(x)$ is the area in the point x whose normal vector is ω, and σ_x is the solid angle by placing the unit sphere's origin in the point x. The measure units of the radiance are $[J \cdot m^{-2} \cdot sr^{-1} \cdot s^{-1}]$.

The *energy flow* Φ is defined as the energy per time unit

$$\Phi = \frac{dQ}{dt}$$

and it is measured in watts $[W = J \cdot s^{-1}]$. It can be proved (Kajiya [Kaj86]) that the previously-described system tends to a *steady-flow state* as time goes by. In these systems, the flow is time-independent. Due to the fact that the velocity of the particles is very high, we assume that the system reaches the steady-flow state almost instantaneously and the system remains in that state during rendering.

As usual in Computer Graphics texts, the radiance is not described in terms of the energy but flow. So, the equation 6.1 can be equivalently rewritten as

$$L(x \to \omega) = \frac{d\Phi}{dA^{\omega}(x) \, d\sigma_x(\omega)}$$

The radiance is, hence, defined as the energy flow per area unit per solid angle unit, and measured in $[W \cdot m^{-2} \cdot sr^{-1}]$.

Notice that the restriction on the fact that the point x has to be on some surface can be removed. Now, it is possible to calculate the incoming or outgoing radiance in any point in the space if a plane is placed on that point and the energy passing thought its normal is calculated. So, we assume there exists —incoming or outgoing— radiance on every point in the space and at any direction.

Radiance is the proper magnitude to measure a particle beam because its value is preserved along a straight line. Mathematically, this property is expressed as the outgoing radiance in a point y at a direction ω is the same as the incoming radiance in other point x at the direction $-\omega$

$$L(x \leftarrow -\omega) = L(y \rightarrow \omega) \tag{6.2}$$

for any pair of mutually visible points, where ω is the direction from y to x.

6.4. Light Transport Problem

Our aim is to obtain realistic images based on the above particle model. So, we have to relate the emission light to the light reaching the camera. The outgoing radiance, only due to scattering, in a surface point is the integral of the incoming radiance multiplied by the BRDF f_r in that point

$$L(x \rightarrow \omega_o) = \int_{\omega_i \in \mathcal{H}_x} f_r(\omega_o, x, \omega_i)\, L(x \leftarrow \omega_i)\, |N_x \cdot \omega_i|\, d\sigma_x(\omega_i) \tag{6.3}$$

$\mathcal{H}_x$ is the hemisphere on the point x and the normal N_x points to its pole. The equation 6.3 only accounts the incoming radiance but not the radiance generated on the surface itself. So, the emission radiance, $L_e(x \rightarrow \omega_o)$, has to be added to obtain the total outgoing radiance. The function L_e is defined as part of the scene.

On the other hand, we will relate the same function in the equation 6.3 to obtain a recursive equation, i.e. we will relate outgoing radiance to outgoing radiance. So, we use the property asserting the radiance is preserved along a straight line (equation 6.2) and we derive the *rendering equation in directional form*:

$$L(x \rightarrow \omega_o) = L_e(x \rightarrow \omega_o) + \int_{\omega_i \in \mathcal{H}_x} f_r(\omega_o, x, \omega_i)\, L(z \rightarrow -\omega_i)\, |N_x \cdot \omega_i|\, d\sigma_x(\omega_i) \tag{6.4}$$

where $z = rc(x, \omega_i)$ is the *ray casting* function, which returns the nearest intersection point from x at the direction ω. The final value I_j of the pixel j is obtained by integrating the irradiance entering the camera multiplied by its important function $W_e^{(j)}$

$$I_j = \int_{x \in \mathcal{C}} W_e^{(j)}(x) E(x)\, dA^{N_x}(x) = \int_{x \in \mathcal{C}} \int_{\omega \in O_x} W_e^{(j)}(x) L(x \leftarrow \omega) |N_x \cdot \omega|\, d\sigma_x(\omega)\, dA^{N_x}(x) \tag{6.5}$$

where $\mathcal{C}$ is the surface of the film and O_x are the directions whose origins are x and pass through the camera opening. For the sake of simplicity, we assume the film is part of the scene, $\mathcal{C} \subset \mathcal{M}$.

The solutions of the equations 6.4 and 6.5 are also the solutions of the light transport problem and, therefore, the rendering of a physically-based image. However, the rendering equation in directional form (equation 6.4) only provides a partial, local insight of the contributions of the light-emitting objects to the final image. So, it is necessary to unfold the recursive equation 6.4 till the outgoing radiance meets the emission radiance.

An alternative way to understand the light transport is to consider each light beam as a whole beginning on a light and finishing in the camera. We define a *path* with length n as a finite

sequence of points $x_0 \dots x_n$ laying on the surfaces of the objects, $x_i \in \mathcal{M}$. The *contribution* of each path is the irradiance reaching the first point, x_0, only from the last one, x_n. The final value for a pixel is the integration over the contributions of every path. That equation is called *rendering equation in path form*.

In order to derive this equation, we have to previously derive the *rendering equation in three-point form* (E. Veach [Vea98], pages 220–222), which relates the irradiance entering the point x from the point y by means of another point z

$$L'(x \leftarrow y) = G(x,y) \int_{z \in \mathcal{M}} f_r(\omega_o, y, \omega_i)\, L'(y \leftarrow z)\, G(y,z)\, dA^{N_z}(z) \tag{6.6}$$

where the vectors ω_i and ω_o are directions of the incoming and outgoing radiance, obtained with the points x, y and z. The term G is called *geometric term* and it is defined as

$$G(x,y) = \frac{|N_x \cdot (y-x)|\ \ |N_y \cdot (y-x)|}{||x-y||^4} V(x,y)$$

and V is the *visibility term*, which is 1 is both points are mutually visible and 0 otherwise.

From the equation 6.6 we can obtain the total irradiance of a set of paths with length n by unfolding the equation n times. For example, the radiance entering the point x_0 from every path with length 3 ($x_0 x_1 x_2 x_3$) is

$$\begin{aligned} E(x_0) = \int_{x_1 \in \mathcal{M}} L'(x_0 \leftarrow x_1)\, dA^{N_1}(x_1) = \int_{x_1 \in \mathcal{M}} G(x_0, x_1) \int_{x_2 \in \mathcal{M}} f_r(\omega_1, x_1, -\omega_2) G(x_1, x_2) \\ \int_{x_3 \in \mathcal{M}} f_r(\omega_2, x_2, -\omega_3) G(x_2, x_3) L_e(x_3 \to \omega_3)\, dA^{N_3}(x_3)\, dA^{N_2}(x_2)\, dA^{N_1}(x_1) \end{aligned} \tag{6.7}$$

In general, the contribution f of a path with lenght n obtained by the equation 6.7 is defined as

$$f(x_0 \dots x_n) = \left(\prod_{i=1}^{n-1} G(x_{i-1}, x_i) f_r(\omega_i, x_i, -\omega_{i+1}) \right) G(x_{n-1}, x_n)\, L_e(x_n \to \omega_n) \tag{6.8}$$

where each ω_i is the unit vector from x_i to x_{i-1}.

An image is rendered from all the energy entering the camera. Let Ω_n be the set of all paths with length n entering the camera from any light, namely paths whose first point x_0 is on the camera film, the second one x_1 is a visible point from the camera, the last one x_n is on a light, and the remaining ones are on the surfaces of the objects of the scene. The set of every path with any length is defined as

$$\Omega = \bigcup_{n \geq 1} \Omega_n \quad \text{where} \quad \Omega_n = \overbrace{\mathcal{M} \times \cdots \times \mathcal{M}}^{n+1 \text{ times}}$$

The rendering equation in path form is now established as

$$I_j = \int_{\pi \in \Omega} W_e^{(j)}(\pi)\, f(\pi)\, d\mu(\pi) \tag{6.9}$$

where

$$d\mu(x_0 \dots x_n) = dA(x_0) \dots dA(x_n)$$

and f is the contribution of each path (equation 6.8). The equation 6.9 represents in short the contribution of all paths to the pixel j. Any method used to solve or to approximate the above

equation for every pixel provides a two-dimensional matrix of colours, i.e. an image. Because these equations simulate the behaviour of light in the real world, the rendered images have a photo-realistic appearance. However, the integrals of the equations 6.9 are difficult to be resolved, so it is usually approximated by numerical methods. The most usual method is Monte Carlo, based on the evaluation of random paths.

6.5. Integration by Monte Carlo

Consider the next integral

$$I = \int_{x \in \Omega} f(x)\ d\mu(x)$$

We will approximate I by the probabilistic method called Monte Carlo. It is said that the random variable $\hat{I}$ is an unbiased estimator of I if it satisfies that its expectation is the desired value

$$E[\hat{I}] = I$$

If $X \sim p$ is a random variable that takes values over the integration domain, and p is the *probability density function* (*pdf*), then the random variable

$$\hat{I} = \frac{f(X)}{p(X)}$$

is an unbiased estimator of I. This can be easily proved by directly applying the definition of expectation

$$E[\hat{I}] = E\left[\frac{f(X)}{p(X)}\right] = \int_{x \in \Omega} \frac{f(x)}{p(x)}\, p(x)\ d\mu(x) = I$$

Also, it must be satisfied that $p(x) \neq 0$ when $f(x) \neq 0$, for all x over the integration domain. Thereby, it is sure that the random variable X does not "cut off" any value in the integration domain.

Let $\hat{I}_1, \ldots, \hat{I}_N$ be N unbiased estimators of I, therefore $E[\hat{I}_i] = I$ for each $\hat{I}_i$, and let

$$\hat{F}_N = \frac{1}{N} \sum_{i=1}^{N} \hat{I}_i$$

be the arithmetic mean of these N estimators. So, the arithmetic mean is also an unbiased estimator of the integral I

$$E[\hat{F}_N] = E\left[\frac{1}{N} \sum_{i=1}^{N} \hat{I}_i\right] = \frac{1}{N} \sum_{i=1}^{N} E[\hat{I}_i] = I$$

Therefore, the process of approximate an integral by Monte Carlo is as follows. First of all, N points are randomly taken throughout the integration domain. This process is called *sampling*, and each point is called a *sample*. After that, the function f is evaluated on those samples and they are divided by their pdf. The arithmetic mean of all of these estimators is an approximation of the integral. The higher the value N is, the nearer the value $\hat{F}_N$ is with respect to the integral I. The *Law of Large Numbers* proves this fact, and it is usually written as

$$P\left(\lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} \hat{I}_i = I\right) = 1$$

A way to quantify the error between an unbiased estimator $\hat{I}$ and the integral I is to analyse the *Mean Squared Error* (*MSE*), also called *variance*

$$E[(\hat{I} - I)^2] = V[\hat{I}] = E[\hat{I}^2] - E[\hat{I}]^2 = \int_{x \in \Omega} \frac{f^2(x)}{p(x)} \, d\mu - I^2$$

The square root of the variance is called *standard deviation* $\sigma[\hat{I}]$.

If $\hat{F}_N$ is the arithmetic mean of N unbiased estimators and these estimators have the same variance, then the diminution rate of the variance is N

$$V[\hat{F}_N] = V\left[\frac{1}{N}\sum_{i=1}^{N} \hat{I}_i\right] = \frac{1}{N} V[\hat{I}]$$

and the diminution rate of MSE is $\sqrt{N}$

$$\sigma[\hat{F}_N] = \sigma\left[\frac{1}{N}\sum_{i=1}^{N} \hat{I}_i\right] = \sqrt{V\left[\frac{1}{N}\sum_{i=1}^{N} \hat{I}_i\right]} = \frac{1}{\sqrt{N}} \sigma[\hat{I}]$$

The lower the variance is, the better approximations of the integral can be obtained with fewer samples.

6.6. Approximation of the Rendering Equation by Monte Carlo

In section 6.4 we saw that the problem of rendering an image from a three-dimensional scene can be equivalently expressed as several integrals. So, calculating the colour of each pixel is now the approximation of the rendering equations for each pixel. Although the integrals of each pixel can be expressed in several forms, we will preferably use the path form (equation 6.9).

The method of Monte Carlo has two advantages with respect to numerical quadrature methods. On the one hand, its convergence rate is always $1/\sqrt{N}$, where N is the number of samples, irrespective of the dimension of the integration domain. On the other hand, the integration domain has infinite dimensions (equation 6.9) because there is no limit for the length of paths. For these reasons, Monte Carlo is the favourite method for approximating the rendering equations.

6.6.1. Sampling of Random Paths

Several algorithms have been developed to sample random paths. In short, the differences among them are the way the random points of the paths are sampled. Here, we present the most representative ones.

J. Kajiya [Kaj86] proposed a sampling method where the first point of each path is directly chosen from the camera. Each subsequent point x_{i+1} is obtained by tracing a ray from the last sampled point x_i. This algorithm is called *Path Tracing*. The paths sampled with Path Tracing traverse the scene in the reverse order of the light particles.

The method by Dutre et al. [DLW93] samples paths from lights to the camera. The name of this sampling algorithm is *Light Tracing*. In order to improve this method, the authors proposed two optimizations. The first one, a ray is traced from each point of the paths to the camera to test the visibility. If there is a hit, a full path is finished and it contributes to the final image. The second one, a set of points is sampled around each point and they are again traced to the camera.

Lafortune and Willems [LW93] presented a new way to generate paths. Two subpaths are sampled together, one from the camera and other from the light. Pairs of points of both subpath are connected to form a bunch of finished paths. This sampling algorithm is called *Bidirectional Path Tracing* (*BPT*). Veach and Guibas [VG94] proposed a new way to weight the completed path by taking into account that every path can be sampled by several different ways.

Veach and Guibas [VG97] applied the Metropolis-Hastings sampling for the generation of paths (*Metropolis Light Transport* or *MLT*). Thus, when an initial full path has been sampled, it is mutated to get another. This new path is accepted if its contribution is higher than the previous one. If its contribution is lower, it can be randomly accepted or rejected. Cline et al. [CTE05] varied this method to distribute the energy carried by paths already generated with Path Tracing.

6.6.2. Russian Roulette

Arvo and Kirk [AK90] brought the *Russian roulette* algorithm from the simulation of neutron transport to Computer Graphics. During the sampling of a path, this method is used to probabilistically take the decision to continue extending a path or to terminate it.

Chapter 7

Graphics Processing Unit

7.1. Introduction

Graphics Processing Units or *GPUs* have been coprocessors traditionally responsible for implementing in hardware the stages of *Graphics Pipeline* algorithm. The current GPUs are fully-programmable parallel hardware and their architecture design has been motivated by this algorithm. Here, we present a brief historical introduction to GPUs. A more complete introduction can be found in the book by Kirk and Hwu [KH10].

The Graphics Pipeline is an algorithm responsible for transforming the triangles of the objects in the scene into a two-dimensional image. Input triangles are defined by three vertices and they are sent by the application to GPU (one by one or in batches). The output image is an array of colours (and, perhaps, other data such as *depth* or *normal vectors*) obtained by the projection of these triangles on the camera film.

From the input of the triangles into the Graphics Pipeline until the final image is completed, the data are transformed by several stages sequentially executed. These stages are mainly grouped into two phases: the *vertex stage* and the *raster stage*. The vertex stage is responsible for calculating the final position of each vertex on the film by multiplying its initial position by affine and projection matrices. In addition, you can add a normal per vertex to apply an illumination model to get a colour per vertex. The raster stage is responsible for calculating the final colour of pixels that are occupied by the projected triangles by interpolating the position and colour of their vertices. Since the latter stage gives each pixel a colour, the Graphics Pipeline is also known as *Raster Algorithm*.

Since the early 80s to late 90s, all stages of the Graphics Pipeline have been implemented in dedicated hardware known as Graphics Processing Unit. However, the functionality of these stages was fixed and could only be barely customized through certain parameters.

Eventually, new visual effects appeared which could be implemented with the Graphics Pipeline. Thus, the vertex stage became increasingly more programmable and each programmer could execute his/her own program on that stage. These programs, which are responsible for modifying the vertices of the triangles, are called *vertex shaders*. Later, raster stage also became a programmable stage, and programs could be embedded in the hardware to manipulate the information associated with each pixel. These programs are called *pixel shaders* or *fragment shaders*. In 2001, NVidia GeForce 3 introduced these features and subsequent GeForce 6 and 7 series continued their development. Despite increased programmability, each of these two stages were implemented by different hardware within the chip.

With the advent of GeForce 8800 in 2006, it was presented the so-called *Unified Architecture*, i.e. vertex and raster stages ran on the same functional units. In addition, some details

of the underlying parallel architecture were exposed so that developers could have more control over the hardware. In order to program this new hardware, it was also presented a high-level language similar to C. This set of features was called *CUDA* (which stands for *Compute Unified Device Architecture*), although it is common to use this name only to talk about the programming language.

7.2. Architecture

7.2.1. Overview

The previous-mentioned scheme of the Graphics Pipeline motivated the development of the GPU architecture. First, the position and colour of each vertex does not depend on other vertices, i.e. all vertices processed in the vertex stage are mutually independent. Therefore, the programmer writes a vertex shader only for one vertex, and the program will be applied on all of them independently. This property is known as *data-independence* and motivated the SIMD (which stands for *Single Instruction on Multiple Data*) architecture of GPUs. In this architecture, several functional units (*cores*) execute the same instructions on different data in parallel. In the raster stage, the same thing happens and the fragments are data-independence.

Second, the *framebuffer* is a region of memory which stores the image rendered by the Graphics Pipeline. The framebuffer includes several buffers, some of them are the *color buffer* (where the colour of the pixels are kept) or the *z-buffer* (which stores the depth of each pixel). The framebuffer is usually read from or written to when a triangle is rendered, by following the stream programming model. This means that memory accesses are performed on adjacent positions and the data are not immediately reused. This led to provide memory systems with higher bandwidth than those in CPUs.

Third, data-independence also motivated the use of *multithreading* to hide the memory *latency*. Latency is the time lapse (in clock cycles) that data take to be available when performing a memory access. Among all the GPU memories, the off-chip memory, called *global memory*, is the slowest. Therefore, when a thread reads from global memory, it will be idle many clock cycles waiting for the data to arrive. During that time, other threads are dispatched, which prevents the processor to be idle. If the processor is not idle at any time, it is said that the memory latency has been hidden.

Fourth, unlike the framebuffer, the use of textures does have a good temporal locality, i.e. if a texel of a texture is retrieved, it is very likely to be retrieved later again. This motivated the use of read-only cache for texture accesses.

7.2.2. Details

The architecture of GPUs has changed since NVidia presented the first GPU with CUDA: the GeForce 8800. In this section we will give some details about one of the latest architectures, called *Fermi* [Nvi09]. GPUs have recently appeared with a new architecture, called *Kepler*. In general, the architectures of the GPUs are specified by two version numbers, called *computer capability* (abbreviated as *cap.*). The first cap. to appear was 1.0 and at the time of writing these pages it is the 3.5 [NVia], corresponding to Kepler. Until otherwise stated, the architectural details that are given in this section correspond to cap. 2.x, i.e. Fermi.

A GPU consists of several *SMs* (which stands for *Streaming Multiprocessors*), and its number depends on the model. Each SM is composed mainly of thread schedulers, bank of registers, shared memory, caches and an array of functional units (Figure 7.1).

The array of functional units consists of 32 cores in cap. 2.0 and 48 in cap. 2.1. Each

Figure 7.1: Scheme of one SM on a GPU with cap. 2.0.

core consists of integer and real arithmetic/logic units. Additionally, there are 16 load/store units (LD/ST) and 4 special units (SFU), the latter ones are responsible for performing mathematical functions such as sine or square roots. The number of cores have been incremented along with the cap. For example, there are 8 cores in cap. 1.0, 32 in cap. 2.0 and 192 in cap. 3.0.

The size of the bank of registers is 32K (= 32768 registers). Each register is an *on-chip*, quick-access 32-bit memory where each thread saves its local information. A register cannot be shared by multiple threads and does not belong to any address space (i.e. a register cannot by accessed by an address).

Shared memory is also an on-chip memory. Its accesses are much faster that global memory, although slower than registers. As its name suggests, this memory is common to several threads and it is devoted to exchange information between them. Shared memory consists of several *memory banks*. Data in shared memory are distributed among these banks so that sequential 32-bit sections fall into sequential banks. Each bank can only provide a 32-bit datum during a request, so that your requests will be serialized when multiple threads access the same bank with different addresses. The cap. 1.x has a 16KB shared memory distributed in 16 banks, whereas in cap. 2.x and 3.x it is available up to 48KB, organized in 32 banks.

The scheduler is responsible for managing the threads in each multiprocessor. Threads are clustered into batches of 32 threads, called *warps*. It is always sure that the operations of the 32 threads of a warp are executed at the same time. The maximum number of threads that can reside on a multiprocessor is 1536 (= 48 warps) in cap. 2.0.

The programs executed on the GPUs are called *kernels*. During the execution of a kernel, the next instruction of a warp is possible not to be ready to run, e.g. if there is a read-after-write hazard on a register, if a datum is not available because it has not arrived from memory, or if a warp is waiting for other warps in a barrier. If a thread has operations ready to be executed, it is called an *active* thread. If all of the threads of a warp are active, the warp is called active. The scheduler is responsible for stopping non-actives warps and dispatching active warps from among the available ones. As mentioned above, this technique is known as multithreading.

Memory transferences are always performed in batches called *transactions*. According to the access pattern, these transactions are of 32, 64 or 128 bytes in the cap. 1.2 and 1.3. In cap. 2.0

and higher, accesses from/to global memory are made through two caches. Each SM has one L1 cache and its size is 48KB at most. L2 cache is common to all the SMs and its size is 768KB. From cap. 2.0 and higher, transactions are of 32 or 128 bytes, whether L1 cache is disabled or not, respectively,

There are also read-only caches in the GPU. Specifically, global memory can be read through texture caches. In cap. 1.x there is a shared texture cache per 2 or 3 SMs, whereas since cap. 2.0, each SM has its own texture cache.

7.3. Programming Model

Before the advent of CUDA architectures, GPUs could also be used as high-performance general-purpose parallel processors. However, GPU programming was much more complicated because it had to be done via the graphics API, like OpenGL. This way of GPU programming was the beginning of a new area of research known as *GPGPU* (which stands for *General-Purpose Computing on Graphics Processing Units*).

The classic way in GPGPU programming was firstly accomplished by encoding the data into the RGB colours of textures. Subsequently, the program is written in a pixel shader. Finally, a square in the window that occupied as many pixels as output is drawn. The pixel shader is executed on each pixel of the screen, calculating the desired output, and the results are written back into the framebuffer.

This programming method had several drawbacks. First, it was necessary to know the graphics API to encode the data properly. Therefore, programmers had to know concepts from the world of graphics, such as "fragment" or "texture", which could have nothing to do with the problem to solve. Second, many architectural details were not known, so code optimization was impossible. Finally, random writings were not possible as each fragment is bound to a single location in the framebuffer.

With the advent of CUDA, GPU programming became much easier, since programming is performed in a C-like language. In addition, many architectural details have been revealed and the programmers have more control over performance of applications. The programs written in this language are called *kernels*. The role of the programmer is to write source code that a single thread executes. When launching the kernel, the number of threads is specified and they all run the same instructions. The remaining issues are automatically managed by the GPU.

The threads are grouped into a two-level hierarchy. First, as mentioned above, consecutive groups of 32 threads are put together into warps, whose instructions are executed simultaneously. This makes warp-level barriers unnecessary. However, the disadvantage is that divergences within warps originate stalls. The next level of the hierarchy is the *block*. The decision on how many warps contains a block is a responsibility of programmers. The idea behind blocks is that their threads collaborate to solve a problem since shared memory and barriers are only visible to these threads. Thus, the threads in a block can exchange information or maintain global data per block in shared memory. The set of all blocks and, therefore, all threads, are called *grid*.

The collaboration among threads of different blocks cannot be performed within the same kernel due to the absence of global barriers. However, the completion of a kernel and launching another have the same effect as a global barrier, although some control has to be moved to CPU, which complicates programming.

The number of threads per block and the number of blocks per grid are specified during the launching of a kernel. At that time, the resources of each SM are allocated: registers are distributed among threads and shared memory among blocks. This procedure sets the maximum number of blocks that can simultaneously reside in each SM, whose threads are used for multithreading. The ratio between the number of threads assigned to a multiprocessor and the maximum allowed by

the hardware is called *occupation* of the GPU. Ideally, if each SM has the maximum number of threads, the occupancy ratio is 100%. However, this is difficult to reach in practice due to the requirements of programs.

It is usual not to have enough resources on the GPU to run all the blocks in parallel. Blocks whose resources are not available have to idle. When the execution of all the threads in a block finishes, the resources of that block are freed and a waiting idle block demands them. In architectures before Fermi, the block allocation was performed at the beginning of the execution, although in the latest GPUs each block is executed whenever resources are available.

Chapter 8

Acceleration Structures

8.1. Introduction

The algorithm that finds the nearest intersection point of each ray with the scene is common to all ray tracing algorithms. Let r be a ray defined by its origin o and its direction d. It is said that the ray r *intersects* the object *obj* of the scene if the point

$$o + t_{hit} \cdot d$$

belongs to *obj*, for some positive value $t_{hit} > 0$. The value t_{hit} is called *intersection time* of the ray r with the object *obj*. We define the set of all intersections I_r of the ray r with all the objects in the scene as

$$I_r = \Big\{ t_{hit} \in \mathbb{R} \mid (t_{hit} > 0) \wedge (obj \in Scene) \wedge (o + t_{hit} \cdot d \in obj) \Big\}$$

The *nearest intersection point* t_{min} for the ray r with the scene is defined as

$$t_{min} = min\big(I_r\big)$$

A ray is possible to intersect no object of the scene, thus I_r is an empty set. In that case, we say that the nearest intersection time of a ray is ∞, which indicates that the ray has left the scene.

Obtaining the nearest intersection point is one of the slowest parts of ray tracing algorithms. Therefore, any attempt to accelerate this stage boosts the overall performance. The naive method to obtain the nearest intersection point for a ray, called *brute force method*, sequentially tests the intersection of that ray with all the objects of the scene. Unfortunately, this method is not feasible in practice due to the large amount of ray-object intersections involved during rendering.

To accelerate the obtaining of the nearest intersection point, data structures, called *index structures* or *acceleration structures* (AS hereinafter) have been developed. These structures decide that some sets of objects will not intersect with a certain ray. Thus, it will not be necessary to test the intersections with those objects and they can be safely discarded. Using these structures, the stage where the nearest intersection points are found is turned into the stage where each ray *traverses* the AS. Therefore, this stage is called *traversal*. The traversal of a ray through the AS is composed of a sequence of steps where each ray decides which parts of the scene to *visit* and which parts to *rule out*. So, a large number of intersection tests are precluded.

8.2. Bounding Volume Hierarchy

A *bounding volume* is an object with finite volume that can be intersected by a ray. Spheres and cylinders are examples of these volumes but *AABB* (which stands for *Axis-Aligned Bounding Box*, or simply *box*) are the most usual. These boxes are composed of six pairwise parallel planes, and also parallel to the main axes, i.e. the planes are $x = cte$, $y = cte$ or $z = cte$.

Notice that there can be infinitely many intersection points between a ray and a box. Let *entry point*, or t_{entry}, be the nearest intersection point of the ray with the box, and let *exit time*, or t_{exit}, similarly be the farthest intersection point.

A *BVH* (which stands for *Bounding Volume Hierarchy*) is, typically, a binary tree where each node has attached a bounding volume. The leaf nodes also have attached a list of references to triangles. The main feature of a BVH is that the volume attached to each internal node encloses the volumes of all of its children. If the node is a leaf, then its volume encloses all triangles referenced by that leaf. This feature is essential because if there is not ray-box intersection, then it is sure that there is not intersection of the ray with any of its children. Thus, that entire sub-tree can be discarded during traversal.

8.3. KD-Tree

A KD-Tree is a binary tree. Each leaf contains a list of references to triangles, similar to BVHs. However, each inner node contains an axis-aligned plane, i.e. a plane whose normal can only be $(1, 0, 0)$, $(0, 1, 0)$ or $(0, 0, 1)$. A plane of this kind is specified by a dimension, $k_{pl} \in \{x, y, z\}$, and a position, p_{pl}, and divides the space associated with the node into two parts.

In a KD-Tree, it holds that if the coordinates of the vertices of a triangle in the dimension k_{pl} are less than p_{pl}, then that triangle will be referenced by a leaf of its left child. Similarly, if the vertices are greater, the triangle will be referenced by the right child. If the plane intersects the triangle, then the triangle is referenced by the two children.

Let box_{scene} be the tightest box that encloses the whole scene. Each plane of the KD-Tree divides box_{scene} into two parts, where each part is another box in turn. Thus, a volume can be implicitly defined for each node of the KD-Tree, which corresponds to box_{scene} but cut off by all the planes of its ancestor nodes. Similarly, triangles referenced by a leaf correspond to the triangles intersected by the implicit volume of that leaf.

In a KD-Tree, the traversal finishes when the first intersection point with a triangle is found because, unlike BVHs, there is no overlapping between the volumes of sibling nodes. Therefore, a depth-first traversal ensures that ray-triangle intersection corresponds to the nearest intersection point.

8.4. Acceleration Structure Construction

Construction of ASs for a scene is the process of building a BVH or a KD-Tree which fulfills the above-described characteristics and which references all the triangles of the scene. It is obvious that there are many ways to build acceleration structures for the same scene, although not all of them exhibit the same performance during traversal. At present, the best experimental results have been achieved by following the *top-down* construction by MacDonald and Booth [MB90] and by using *SAH* (stands for *Surface Area Heuristic*) by Goldsmith and Salmon [GS87].

In order to build *good* acceleration structures, it is necessary to define the *cost* on these structures. Thus, a good construction algorithm receives the list of triangles in the scene as input and returns the structure at lowest cost.

We define the cost of a hierarchical structure as the value $|ray_{root}| \cdot cost(root)$, where *root* is the root node of the tree, ray_i is the set of the rays that intersect the node i during rendering, i.e. during the traversal, and *cost* is the average cost per ray of the structure. The function *cost* is defined recursively as

$$cost(n) = \begin{cases} C_t \cdot Tri(n) & \text{if } n \text{ is leaf} \\ C_i + \frac{|ray_l|}{|ray_n|} cost(l) + \frac{|ray_r|}{|ray_n|} cost(r) & \text{if } n \text{ is inner} \end{cases} \tag{8.1}$$

The nodes l and r are the left and right children, respectively, of the node n, and $Tri(i)$ is the number of triangles in the leaf i. Notice that $|ray_l| + |ray_r| \geq |ray_n|$, since a ray is possible to traverse both nodes. C_t is the cost of ray-triangle intersection and C_i is the cost of ray-inner node intersection.

8.4.1. Surface Area Heuristics

The number of rays used to render a scene is huge. Therefore, we can approximate the ratio $\frac{|ray_l|}{|ray_n|}$ as the conditional probability $P(l|n)$ for a ray to intersect the node l given it has intersected the node n, i.e. $\frac{|ray_l|}{|ray_n|} \approx P(l|n)$, and similarly for the right child, $\frac{|ray_r|}{|ray_n|} \approx P(r|n)$.

In order to derive the probability $P(l|n)$ with no knowledge about either $|ray_l|$ or $|ray_n|$, we will use *geometric probability*, which is determined as a ratio of geometric measure functions. To easily obtain a measure function for rays, we will accept these three assumptions:

1. Rays begin out of the scene.
2. Rays are not blocked by objects of the scene during traversal. Therefore, they finish out of the scene.
3. Ray distribution is uniform.

Although these assumptions are not realistic for the rays used during rendering, they are useful to derive the intersection probability of each node of the AS, as we will see.

Let $\mathcal{R}$ be the set of all rays that can be used during rendering. Similar to Chapter 6, we define the direction ω of a ray as the vector that begins at the coordinate system origin and finishes at a point on the surface of the unit sphere $\mathcal{S}^2$. For a direction ω given, we assume that the origin o of the ray is on a plane perpendicular to that direction Π_ω (Figure 8.1). The position of the plane Π_ω is not important as long as it is out of the scene. In addition, the plane Π_ω has to be unique for each direction ω, which avoids ambiguities during the specification of rays. Thus, the set of rays $\mathcal{R}$ is defined as

$$\mathcal{R} = \Big\{ (o, \omega) \mid \omega \in \mathcal{S}^2 \ \wedge \ o \in \Pi_\omega \Big\}$$

Note that every ray in $\mathcal{R}$ fulfills trivially the first of the three assumptions.

Let the probability space be the set of all rays in $\mathcal{R}$ that intersect the scene box

$$\mathcal{R}_{root} = \Big\{ r \in \mathcal{R} \mid r \text{ intersects } box_{root} \Big\}$$

The events of this probability space are the sets $\mathcal{R}_n$, where n is a node of the AS. Let $P(n)$ be the probability of the node n, i.e. the probability for a random ray to intersect its attached box box_n. In other words, it is the probability for a ray to belong the event $\mathcal{R}_n$

$$P(n) = P(\mathcal{R}_n)$$

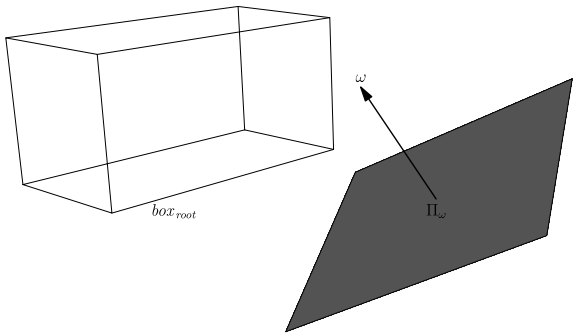


Figure 8.1: Specification of rays under the three assumptions. The origins of all rays with the same direction ω are on the plane Π_ω, which is perpendicular to the direction and it is out of the box of the scene.

The second assumption allows to derive a probability for $P(n)$ ignoring any other box or triangle in the scene, except the box itself and the box of the scene. This implication is due to the fact that we have assumed that no ray can be blocked during traversal, so the probability of intersecting a box is irrespective of the objects around.

However, the second assumption also entails some *symmetries* for the rays of an event $\mathcal{R}_n$. Thus, if a ray whose direction ω intersects box_n, then another ray with direction $-\omega$ also intersects it. In order not to consider this pair of rays multiple times in the probability of a box, we restrict the space of the rays only to those whose directions belong to the canonical hemisphere $\mathcal{H}$

$$\mathcal{R} = \Big\{(o,\omega) \mid \omega \in \mathcal{H} \ \wedge \ o \in \Pi_\omega\Big\}$$

We remark that $\mathcal{H}$ is the unit hemisphere, centered at the origin and whose pole points at $(0,0,1)$. Note that $\mathcal{R}$ is now the set of straight lines in $\mathbb{R}^3$, so we talk about straight lines and rays interchangeably during the derivation of the probability.

The third assumption says that the probability density of rays is constant. To derive the pdf of rays, we define the measure μ for sets of rays. This measure is defined from the surface area A and the solid angle σ as

$$\mu(\mathcal{R}_n) = \int_{r\in\mathcal{R}_n} 1\, d\mu(r) = \int_{r\in\mathcal{R}_{root}} \chi_{\mathcal{R}_n}(r)\, d\mu(r) = \int_{\omega\in\mathcal{H}}\int_{o\in\Pi_\omega} \chi_{\mathcal{R}_n}(r)\, dA(o)\, d\sigma(\omega) \tag{8.2}$$

where $\chi_{\mathcal{R}_n}$ is the characteristic function of the set $\mathcal{R}_n$, i.e. the function is 1 if the ray belongs to $\mathcal{R}_n$, and 0 otherwise. With this measure, the pdf of a ray is as follows

$$p(r) = \frac{1}{\mu(\mathcal{R}_{root})}$$

for every ray $r \in \mathcal{R}_{root}$. The probability of an event is, by definition, the integral of the pdf over all the rays of the event

$$P(n) = P(\mathcal{R}_n) = \int_{r\in\mathcal{R}_n} p(r)\, d\mu(r) = \frac{\mu(\mathcal{R}_n)}{\mu(\mathcal{R}_{root})}$$

Now, given a fixed direction ω, the area enclosing the characteristic function of $\mathcal{R}_n$ is the area of the orthogonal projection of box_n in the direction ω (Figure 8.2), so the equation 8.2 becomes

$$\mu(\mathcal{R}_n) = \int_{\omega\in\mathcal{H}} A\big(proy\_orth(box_n,\omega)\big)\, d\sigma(\omega) \tag{8.3}$$

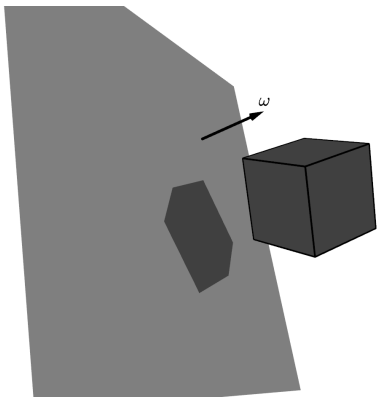


Figure 8.2: Orthogonal projection of a box on the plane Π_ω.

Since box_n is an AABB, only one face of each pair is visible at most. The area of the orthogonal projection of these faces is obtained by multiplying their areas by the cosine of the angle between ω and their normals. Let Δ_x, Δ_y and Δ_z be the areas of each face of each pair, and let θ_x, θ_y and θ_z be the angles between ω and the normals of each face, then the area of the projection is

$$A\big(proy\_orth(box_n, \omega)\big) = \Delta_x|\cos\theta_x| + \Delta_y|\cos\theta_y| + \Delta_z|\cos\theta_z|$$

The normals are $N_x = (1,0,0)$, $N_y = (0,1,0)$ and $N_z = (0,0,1)$, and their opposite vectors, therefore, the orthogonal projection can be expressed by using the dot product, as

$$A\big(proy\_orth(box_n, \omega)\big) = |N_x \cdot \omega|\Delta_x + |N_y \cdot \omega|\Delta_y + |N_z \cdot \omega|\Delta_z = |\omega_x|\Delta_x + |\omega_y|\Delta_y + |\omega_z|\Delta_z$$

If we unfold the equation 8.3, we have

$$\begin{aligned}
\mu(\mathcal{R}_n) &= \int_{\omega\in\mathcal{H}} \Big(|\omega_x|\Delta_x + |\omega_y|\Delta_y + |\omega_z|\Delta_z\Big)\, d\sigma(\omega) \\
&= 4\int_0^{\pi/2}\int_0^{\pi/2} \Big(\Delta_x \sin\theta\cos\phi + \Delta_y \sin\theta\sin\phi + \Delta_z\cos\theta\Big)\sin\theta\, d\theta\, d\phi \\
&= \pi(\Delta_x + \Delta_y + \Delta_z) \\
&= \frac{\pi}{2} SA(box_n)
\end{aligned}$$

where $SA(box_n)$ is the surface area of box_n. Thus, the intersection probability of the node n by a ray under the three above assumptions is

$$P(n) = P(\mathcal{R}_n) = \frac{\mu(\mathcal{R}_n)}{\mu(\mathcal{R}_{root})} = \frac{SA(box_n)}{SA(box_{root})}$$

This heuristics, which estimates the intersection probability of a node as the ratio between surface areas, is called *SAH* (which stands for *Surface Area Heuristics*).

The probability that a ray intersects the node n, given it has intersected its parent node m, is the conditional probability of the event $\mathcal{R}_n$ given $\mathcal{R}_m$

$$P(n|m) = P(\mathcal{R}_n|\mathcal{R}_m)$$

With the above assumptions, if $box_n \subseteq box_m$, then it is also true that $\mathcal{R}_n \subseteq \mathcal{R}_m$. Likewise, the following facts are true

$$\begin{array}{lrcll} (1) & P(n|m) & = & \frac{P(\mathcal{R}_n \cap \mathcal{R}_m)}{P(\mathcal{R}_m)} = \frac{P(n)}{P(m)} & \text{if } box_n \subseteq box_m \\ (2) & P(n|q) & = & \frac{P(n)}{P(q)} = \frac{P(n)}{P(m)}\frac{P(m)}{P(q)} = P(n|m)P(m|q) & \text{if } box_n \subseteq box_m \subseteq box_q \\ (3) & P(n|root) & = & \frac{P(n)}{P(root)} = P(n) & \end{array}$$

8.4.2. Estimation for the Cost of a Structure

Given a partition of the triangle list in a node, an estimation of the cost of that node in the final tree can be calculated by using the above assumptions. Thus, the function $\hat{cost}$, which approximates *cost* (equation 8.1) is

$$\hat{cost}(n) = C_i + C_t \frac{SA(box_l)}{SA(box_n)} Tri(l) + C_t \frac{SA(box_r)}{SA(box_n)} Tri(r) \tag{8.4}$$

where C_t is always 1, and C_i is 1 or 2 depending on whether the AS to build is a KD-tree or a BVH, respectively. This function $\hat{cost}$ is called *estimated cost* or simply *SAH cost*.

The top-down construction algorithm of an AS for a certain scene therefore consists of the test of all possible divisions of the list of the triangles. The SAH cost of the node is evaluated for each possible division as well as the SAH costs of the child nodes are evaluated as if they were leaves. If the minimum of these costs is the cost of the leaf, then the list of the triangles becomes a leaf. If the minimum occurs in a division, then that division will be chosen and the procedure will recursively continue on its two children. The specific details of which divisions are considered are different for KD-Trees and BVHs.

8.5. Specialized Heuristics

The derivation of the Surface Area Heuristics comes from considering the fact that rays can be approximated as uniformly distributed straight lines (section 8.4). Accepting different assumptions for these rays leads to the development of cost estimations other than the original SAH. In Torres et al. [TMGA12] two assumptions have changed. The first one is that all possible directions are not considered, but only those that belong to a certain set. The second one is to assume that not all the rays are equally likely, but some are more likely than others.

The restriction of the set of rays allows the derivation of more specific cost estimates. The ASs built with these new cost estimations are expected to be more efficient for those rays of their corresponding set, although this must be experimentally tested (section 8.5.7).

8.5.1. Spherical-Patch Surface Area Heuristics

One way to restrict the set of rays is to shrink their directions. If we assume the other assumptions, rays can be approximated by straight lines as before. However, direction vectors of these lines will not be points on the whole unit hemisphere, but only on a subset of it. Notice that this new restriction does not affect the ray origins and, hence, they can be placed anywhere, provided that they meet the other assumptions, i.e. they are out of the scene on a plane perpendicular to their directions.

Let $T : [0, \pi] \times [0, 2\pi] \to \mathcal{S}^2$ be the function that transforms from spherical coordinates to Cartesian coordinates

$$T(\theta, \phi) = (\sin\theta\cos\phi,\ \sin\theta\sin\phi,\ \cos\theta)$$

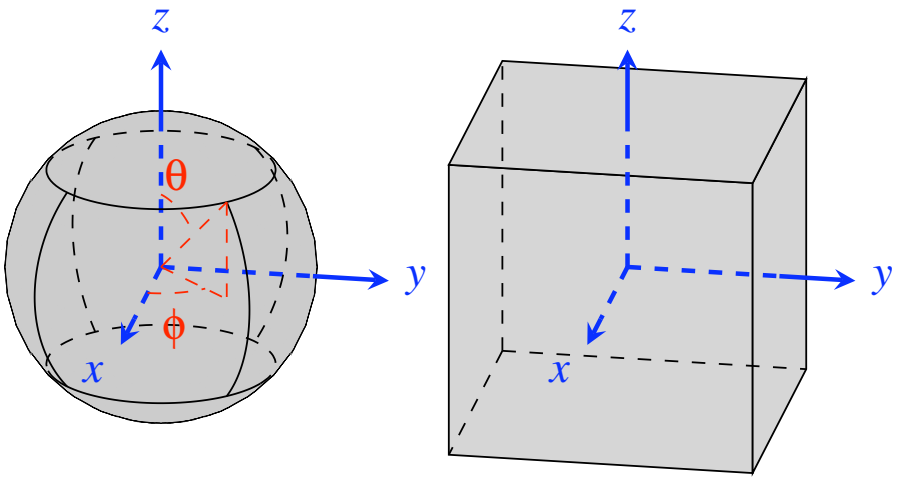


Figure 8.3: Distribution of the spherical patches (on the left) and the cubic patches (on the right). For the sake of clarity, the six patches are shown in both figures, although just three of them are considered to approximate the probability.

Patch	**Bounds (spherical coord.)**		**SPHERE-ORTH**			**SPHERE-OBLI**		
	Θ	Φ	w_x	w_y	w_z	w_x	w_y	w_z
SP_x	$[\theta_0, \pi - \theta_0]$	$[\frac{-\pi}{4}, \frac{\pi}{4}]$	55.04	22.80	22.15	53.47	23.59	22.92
SP_y	$[\theta_0, \pi - \theta_0]$	$[\frac{\pi}{4}, \frac{3\pi}{4}]$	22.80	55.04	22.15	23.59	53.47	22.92
SP_z	$[0, \theta_0]$	$[0, 2\pi]$	22.04	22.04	55.90	22.66	22.66	54.67
Patch	**Bounds (Cartesian coord.)**			**CUBE-ORTH**			**CUBE-OBLI**	
	x	y	z	w_x	w_y	w_z	w_x	w_y	w_z
CP_x	$\{1\}$	$[-1, 1]$	$[-1, 1]$	51.29	24.35	24.35	50.00	25.00	25.00
CP_y	$[-1, 1]$	$\{1\}$	$[-1, 1]$	24.35	51.29	24.35	25.00	50.00	25.00
CP_z	$[-1, 1]$	$[-1, 1]$	$\{1\}$	24.35	24.35	51.29	25.00	25.00	50.00

Table 8.1: Bounds and normalized weights for spherical and cubic heuristics. The values w_x, w_y and w_z are the normalized weights in percentage for the face areas Δ_x, Δ_y and Δ_z, respectively.

Let $\Theta \subset [0, \pi]$ and $\Phi \subset [0, 2\pi]$ be two intervals in the spherical coordinate space. If T is applied on the rectangle $\Theta \times \Phi$, then a spherical section is obtained called *spherical patch* or *SP*. Since rays are approximated as straight lines, the rectangle $\Theta \times \Phi$ has to be chosen so that directions ω and $-\omega$ are not on the same *SP*.

In our paper, three different spherical patches are tested, SP_x, SP_y and SP_z (Figure 8.3 on the left). The intervals Θ and Φ that spawn these spherical patches can be retrieved in Table 8.1. The constant θ_0 is set to $\theta_0 = acos(\frac{2}{3})$ so that surfaces of all patches are equal.

In order to define a measure function for a set of rays and, therefore, to derive the probability for a node n to be intersected, it is enough to restrict the integral domain in the equation 8.2 to one spherical patch. Each spherical patch SP_x, SP_y and SP_z brings a new measure, respectively denoted as $\mu_{orth}^{SP_x}$, $\mu_{orth}^{SP_y}$ and $\mu_{orth}^{SP_z}$

$$\mu_{orth}^{SP_i}(\mathcal{R}_n) = \int_{\omega \in SP_i} A\big(proy\_orth(box_n, \omega)\big)\, d\sigma(\omega) \tag{8.5}$$

where $i \in \{x, y, z\}$. The cost estimations for these measures are called SPHERE-ORTH. The integration of the expression 8.5 results in three different weights, w_x, w_y and w_z, for each surface of each pair of faces of box_n, denoted as Δ_x, Δ_y and Δ_z. So, the conditional probability for the node n to be intersected is

$$P(n) = \frac{\mu_{orth}^{SP_i}(\mathcal{R}_n)}{\mu_{orth}^{SP_i}(\mathcal{R}_{root})} = \frac{w_x\Delta_x + w_y\Delta_y + w_z\Delta_z}{w_x\Delta'_x + w_y\Delta'_y + w_z\Delta'_z}$$

where Δ'_x, Δ'_y and Δ'_z are the surface of each pair of faces of the box_{scene}. Since, the probability is defined as a ratio of measures, those weights can be normalized. The values of the normalized weights w_x, w_y and w_z for every SP can be retrieved in Table 8.1. Notice that the face in front of each SP (e.g. the face Δ_x in SP_x) is given a greater weight than the other faces.

8.5.2. Cubic-Patch Surface Area Heuristics

Another way to specify the direction space (based on a paper by Hunt and Mark [HM08a]) is to use an origin-centered cube (Figure 8.3 on the right). To make the derivation easier, we assume that the minimum and maximum points of the cube are $(-1, -1, -1)$ and $(1, 1, 1)$, respectively. This way, ray directions are specified as vectors that begin at the coordinate system origin and finish on the cube's surface. These vectors have to be normalized to be used as ray directions. Since these directions are points on the cube's surface, a usual measure is the surface area.

The direction space can be shrunk if only the directions that finish on a cubic face are considered. Each of them is called *cubic patch* or *CP*. Similar to SPHERE-ORTH, each straight line could be taken into account twice if they were specified by two opposite-signed directions. So, only one face of each pair of parallel faces can be used as integration domain. For the sake of simplicity, we chose the faces whose constant coordinate is 1, which we call CP_x, CP_y and CP_z (Table 8.1).

According to this partition of the direction space, three measures can be derived, $\mu_{orth}^{CP_x}$, $\mu_{orth}^{CP_y}$ and $\mu_{orth}^{CP_z}$, and three cost estimations are obtained. For example, for $\mu_{orth}^{CP_z}$ the following integral has to be solved

$$\mu_{orth}^{CP_z}(\mathcal{R}_n) = \int_{-1}^{1}\int_{-1}^{1} A\left(proy\_orth\left(box_n, \frac{(x, y, 1)}{\sqrt{x^2 + y^2 + 1}}\right)\right) dx\, dy$$

This heuristics is called CUBE-ORTH and the resulting weights can be retrieved in Table 8.1.

8.5.3. Cos-Weighted Probability

Another proposal of ours regarding the relaxation of some assumptions about rays is to consider that all rays are possible but they are not equally likely. Specifically, we changed the probability density for a ray to be proportional to a power of the cosine of the angle between its direction and a given vector D

$$p(r) = \frac{|D \cdot \omega|^\beta}{\displaystyle\int_{r \in \mathcal{R}_{root}} |D \cdot \omega|^\beta \, d\mu(r)} \tag{8.6}$$

The constant β is a positive real used to raise the probability of those rays whose directions are closer to D. If $\beta = 0$, then the pdf is the same as the original SAH.

The probability of a set of rays can be expressed as a ratio between measures, similarly to previous heuristics. In this case, we will use the following measure μ_{orth}^{D}, which is expressed in

	COS-ORTH			COS-OBLI		
β	w_x	w_y	w_z	w_x	w_y	w_z
1	43.99	28.00	28.00	33.33	33.33	33.33
2	50.00	25.00	25.00	43.99	28.00	28.00
3	54.08	22.95	22.95	50.00	25.00	25.00
4	57.14	21.42	21.42	54.08	22.95	22.95
5	59.55	20.22	20.22	57.14	21.42	21.42
10	67.01	16.49	16.49	65.90	17.04	17.04

Table 8.2: Normalized weights in percentage for the cosine heuristics, taking different values of β. We only present the case for N_x. The others can be obtained by suitably swapping columns.

terms of μ as

$$\mu^D_{orth}(\mathcal{R}_n) = \int_{r\in\mathcal{R}_n} |D\cdot\omega|^\beta \, d\mu(r)$$

Thus, the probability density of the equation 8.6 can also be expressed as

$$p(r) = \frac{|D\cdot\omega|^\beta}{\mu^D_{orth}(\mathcal{R}_{root})}$$

and the probability for a ray to intersect a node n is

$$P(n) = P(\mathcal{R}_n) = \int_{r\in\mathcal{R}_n} p(r)\, d\mu(r) = \frac{\mu^D_{orth}(\mathcal{R}_n)}{\mu^D_{orth}(\mathcal{R}_{root})}$$

We used three vectors for D: $N_x = (1,0,0)$, $N_y = (0,1,0)$ and $N_z = (0,0,1)$. The three measures derived from these vectors, $\mu^{N_x}_{orth}$, $\mu^{N_y}_{orth}$ and $\mu^{N_z}_{orth}$, respectively define three cost estimations, which we will call COS-ORTH. Their normalized weights can be retrieved in Table 8.2. Note that it is only necessary to obtain the weights for one measure due to the way vectors D have been chosen. The other measures are permutations of those already obtained.

8.5.4. Oblique Projection

A straight line can be specified by a direction and an origin. According to section 8.4.1, directions are specified as points on the unit hemisphere, whereas origins are points out of the scene and lying on a plane orthogonal to their directions. An alternative way to specify rays is to assume that origins are all on the same plane, regardless of the ray direction. This fact makes origin measure the area of the oblique projection of the box on this plane (Figure 8.4) instead of the area of the orthogonal projection.

The new measures derived from this ray specification are obtained by following the procedure in Sections 8.5.1, 8.5.2 and 8.5.3, but replacing the orthogonal projection with the oblique projection. We have used the same notation for these heuristics, but replacing the suffix ORTH with OBLI. Thus, we obtain nine new measures, denoted by $\mu^{SP_x}_{obli}$, $\mu^{SP_y}_{obli}$, $\mu^{SP_z}_{obli}$ for SPHERE-OBLI, $\mu^{CP_x}_{obli}$, $\mu^{CP_y}_{obli}$, $\mu^{CP_z}_{obli}$ for CUBE-OBLI, and $\mu^{N_x}_{obli}$, $\mu^{N_y}_{obli}$, $\mu^{N_z}_{obli}$ for COS-OBLI. Tables 8.1 and 8.2 include the resulting weights.

8.5.5. Multiple Specialized KD-Trees

During the rendering of a scene, the rays can take any direction. If only a specialized KD-Tree is used as acceleration structure, those rays whose directions belong to *SP* or *CP* will

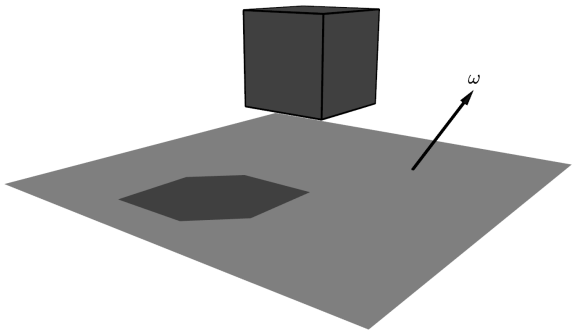


Figure 8.4: Oblique projection of a box on a plane at the direction ω.

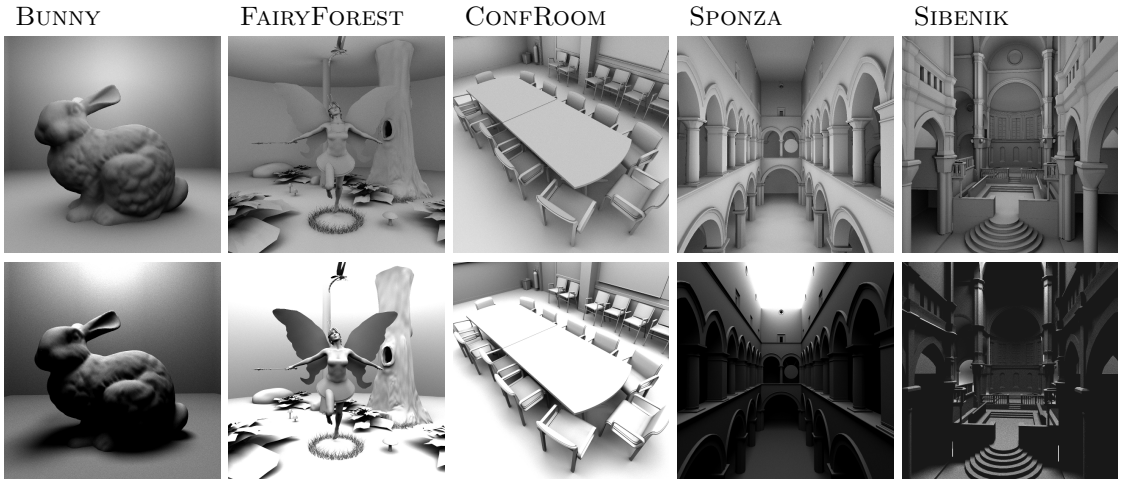


Figure 8.5: Snapshot of the scenes used in our experiments. In the first row there are the scenes rendered with Ambient Occlusion, and in the second row the scenes rendered with Path Tracing.

obtain a faster traversal, whereas rays whose directions do not belong these sets will require more time. So, it is essential to use enough specialized KD-Trees during the rendering to fully cover the direction space. Due to the way the direction space is partitioned in SPHERE and CUBE, only three specialized KD-Trees are required. We call the set of these three KD-Trees a *Multi-KD-Tree*.

The heuristics COS are different because they consider all the possible ray directions. We have also checked that using a single COS-built KD-Tree involves a worse performance during traversal. So, we have used the same solution as SPHERE and CUBE, i.e. the usage of a Multi-KD-Tree composed of three KD-Trees. Each of these KD-Trees is built by setting the vector D to the three axis: $D = (1, 0, 0)$, $D = (0, 1, 0)$ and $D = (0, 0, 1)$.

8.5.6. Methodology for Testing the Multi-KD-Tree

We have implemented a Path Tracing (PT) and an Ambient Occlusion (AO) in CUDA as ray tracing algorithms to test the performance of a Multi-KD-Tree built with the above heuristics. The scenes used in our tests are Bunny, FairyForest, ConfRoom, Sponza and Sibenik (Figure 8.5). The generated images have a resolution of 1024×1024 and every materials are diffuse.

	Primary Rays				
	SAH		Specialized SAH		
Scene	Steps	MRays/s	Best Heuristics	Steps	MRays/s
Bunny	34.04	141.12	SPHERE-ORTH	30.27(11.08)	147.45(4.29)
FairyForest	48.82	101.70	CUBE-ORTH	45.70(6.38)	106.04(4.09)
ConfRoom	38.46	149.19	SPHERE-OBLI	34.78(9.56)	157.52(5.29)
Sponza	37.66	171.75	SPHERE-ORTH	34.52(8.33)	178.78(3.93)
Sibenik	45.03	143.31	CUBE-OBLI	38.67(14.11)	156.83(8.61)
	Secondary Rays				
	SAH		Specialized SAH		
Scene	Steps	MRays/s	Best Heuristics	Steps	MRays/s
Bunny	32.09	36.29	SPHERE-OBLI	30.88(3.78)	37.33(2.78)
FairyForest	51.51	19.36	CUBE-OBLI	49.11(4.64)	20.33(4.75)
ConfRoom	39.83	26.21	CUBE-ORTH	38.81(2.55)	27.57(4.93)
Sponza	41.17	26.14	CUBE-OBLI	39.50(4.06)	28.11(7.00)
Sibenik	48.01	19.66	CUBE-OBLI	45.78(4.63)	21.41(8.14)

Table 8.3: Best results for Path Tracing.

Our system is executed on a Nvidia GeForce 285 GTX with 1GB of RAM memory.

Implementation Details for Path Tracing

In this ray tracing algorithm, only two levels of recursion are accounted: *primary* and *secondary rays*. The algorithm is composed of three kernels: ray generator (RG), traversal (TI) and shading (SH). This Path Tracing is implicit, i.e. no shadow rays are traced from intersection points to lights. To obtain the final image, several iterations of the kernels are necessary.

Implementation Details for Ambient Occlusion

This renderer also considers two level of recursion: *primary* and *shadow rays*. It is also composed of three kernels, similarly to Path Tracing: ray generator (RG), traversal (TI) and shading (SH). In order to render the final image, it is necessary to iterate over shadow rays several times, but only once for primary rays.

Ray Arrangement

The primary rays are stored in the ray array by following the Morton code of pixels in the image. So, the probability to choose the same KD-Tree for adjacent rays is high. However, secondary rays are randomly spawned, therefore adjacent rays are possible to choose different KD-Trees. This fact results in a lot of cache misses at the beginning of TI because the roots of KD-Trees are very far away in the node array. It can be experimentally checked that the number of traversal steps are lesser but the performance is worse.

To solve this problem, a new kernel *Sort* is executed just before the kernel TI of secondary and shadow rays. This kernel puts together the rays according to their KD-Trees. It is implemented on CUDA by using the *radixsort* primitive by CUDPP [HOS+10]. Since there are three values at most, the sorting has to be performed with only the two least-significant bits.

	Primary Rayos				
	SAH		Specialized SAH		
Scene	Steps	MRays/s	Best Heuristics	Steps	MRays/s
Bunny	34.23	78.72	SPHERE-ORTH	30.46(10.99)	81.29(3.16)
FairyForest	49.03	63.44	SPHERE-ORTH	45.60(6.99)	65.09(2.53)
ConfRoom	38.55	87.87	SPHERE-OBLI	34.88(9.52)	94.45(6.96)
Sponza	37.73	112.70	SPHERE-ORTH	34.59(8.31)	117.33(3.94)
Sibenik	47.31	79.41	CUBE-OBLI	40.80(13.75)	84.55(6.07)

	Shadow Rays				
	SAH		Specialized SAH		
Scene	Steps	MRays/s	Best Heuristics	Steps	MRays/s
Bunny	28.91	46.89	SPHERE-OBLI	28.07(2.90)	46.59(-0.63)
FairyForest	42.58	30.80	CUBE-ORTH	40.62(4.59)	31.46(2.10)
ConfRoom	31.28	52.80	CUBE-ORTH	30.77(1.63)	52.98(0.32)
Sponza	34.35	47.02	SPHERE-OBLI	32.96(4.03)	49.03(4.09)
Sibenik	39.55	37.07	CUBE-ORTH	37.94(4.06)	38.79(4.42)

Table 8.4: Best Results for Ambient Occlusion.

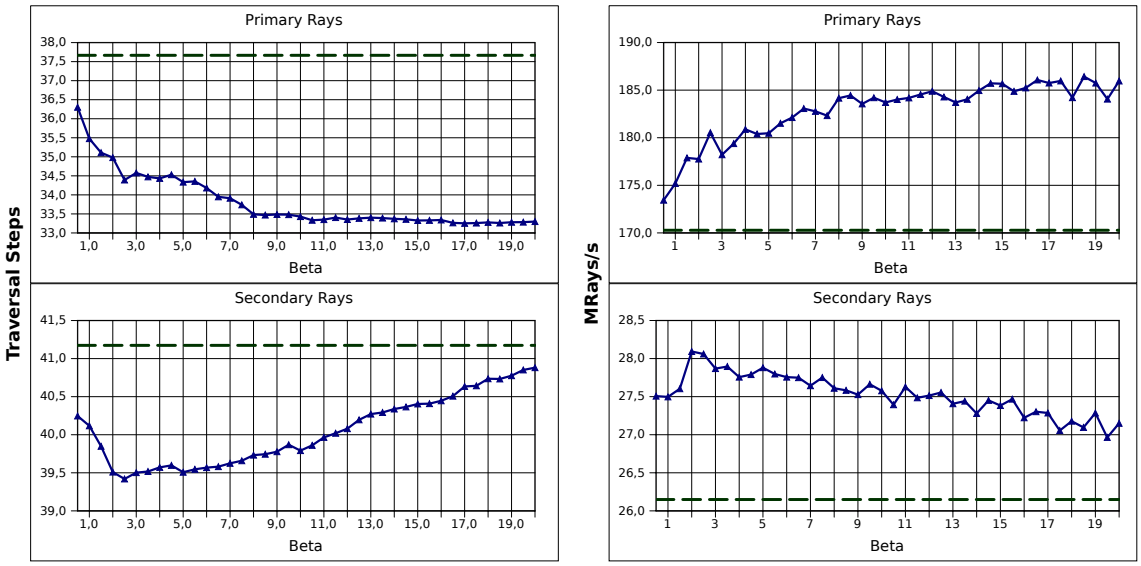


Figure 8.6: Results of COS-ORTH heuristics (blue) for the scenes Sponza, which is rendered with Path Tracing, compared to the same scene with SAH (red).

8.5.7. Results

Tables 8.3 and 8.4 show the best results for Path Tracing and Ambient Occlusion. They are split into two parts. The two "SAH"-tagged columns show the statistics for a SAH-built KD-Tree. The three "specialized SAH"-tagged columns show the statistics for the Multi-KD-Tree with the best performance. The remaining results can be retrieved in Torres et al. [TMGA12].

The column "Steps" is the number of *traversal steps* per ray. The column "MRays/s" is the performance of kernels *Sort* and *TI* measured in million rays per second. Only the kernels *TI* and *Sort* are depicted because they are the most time-consuming kernels. Specifically, *TI* takes

75%-83% of the whole rendering time. The column "Best Heuristics" is the specialized heuristics (SPHERE or CUBE) used for building the Multi-KD-Tree with the best performance. The columns "Steps" and "MRays/s" show in parenthesis the saving percentage with respect to SAH.

As it can be seen in the column "Steps", the rays need fewer traversal steps to obtain their nearest intersection points when a Multi-KD-Tree is used. The saving reaches up to 14.11%. In the column "MRays/s", it is shown an increase of the performance with respect to SAH up to 8.61% for primary rays and 8.14% for secondary rays. However, this increase is not so high as the column "Step" because of cache misses and execution divergences of the CUDA threads.

We have also tested the performance of COS-ORTH and COS-OBLI heuristics. Figure 8.6 shows the results of COS-ORTH for Sponza scene rendered by Path Tracing. The remaining results can be retrieved in Torres et al [TMGA12]. The parameter β (Section 8.5.3) ranges from 0.5 to 20.0 in strides of 0.5. When β is lower than 3.5, the traversal steps decrease and, consequently, the performance increases. The weights resulting from COS-ORTH for these values of β are similar to those from SPHERE and CUBE heuristics. When β increases, the behaviour is scene-dependent.

Chapter 9

Hardware Exploitation

9.1. Introduction

The most straightforward way to accelerate the ray tracing algorithms is to harness the underlying hardware. Several techniques has been used and they can be classified into four groups.

Parallelism. Techniques in this group are oriented to harness all the parallelism in the hardware. This parallelism comes up in two ways. First, chips have several processors, which can independently run different instruction flow. This is known as *MIMD* (stands for *Multiple Instructions, Multiple Data*). The cores of CPUs and SMs of GPUs are examples of this hardware. Second, several function units can run the same instructions but on different data. This is known as *SIMD* (stands for *Single Instruction, Multiple Data*). The SSE instructions of CPUs and cores in SM of GPUs are examples of this hardware.

Division of labour. The whole job is divided into smaller tasks that are distributed throughout the processors. The ray tracing algorithms are embarrassingly parallel if only paths are taken into account. However, the generation of each path must be sequential. These tasks are assigned to the processors and they can require different period of time to complete. So, a dynamic scheduling is more appropriated than a static one.

Memory hierarchy. In these groups, there are techniques intended to optimize the memory system. A good design of algorithms and memory layout allows to increase cache hits and coalesced accesses. In addition, the reuse of the fast on-chip memories, such as registers and shared memory, allows to decrease the number of transactions from memory.

Coding optimization. This last group includes the careful coding of the program. So, if the program requires fewer operations to complete a task or it comsumes less resources, it is possible for the application to be faster.

9.2. Coherence

In order to harness the hardware, the concept *coherence* of a group of rays has appeared throughout the literature. There is no an accurate definition for coherence but they all refer to some common characteristic of rays to take advantage. The most frequent characteristic is to analyse the nodes of the Acceleration Structure that rays intersect during their traversal.

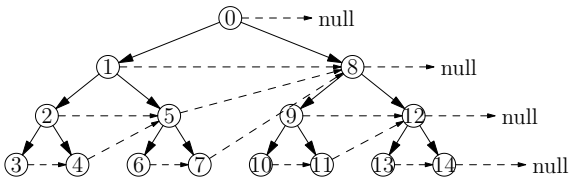


Figure 9.1: Example of a roped BVH. The ropes are shown with dashed lines.

9.3. Ray Casting using a Roped BVH with CUDA

In this section, we present our paper Torres et al. [TMG09a]. This work is about a stackless BVH traversal. Each BVH node is augmented with ropes to guide rays during their traversal. Each rope points to the next node in a preorder traversal of the tree (Figure 9.1). In addition, each group of rays traverses together the BVH in a ray packet. This allows to conform with the restrictive memory-access pattern of GPUs with cap. 1.x to perform coalesced readings.

9.3.1. Ray-Packet Roped-BVH Traversal

Each ray saves the next BVH node in the variable `NR`. In addition, every ray of the packet has the variable `NP` in common, which indicates the next BVH node of the packet. Every ray of the packet collaborates to bring the node pointed to by `NP`, which is accomplished by a coalesced reading when each ray brings some bytes of the node information.

We say that a ray is *active* if its `NR` is equal to `NP`, otherwise, it is *inactive*. Active rays intersect the next packet node `NP` and update its `NR`. Inactive rays do not update `NR`, so they just "wait" in its node `NR` to become active again.

The variable `NP` is updated from the `NR`s of all the rays in the packet. If all the rays in the packet go to the same node, then `NP` is set to that node. If some rays go through the rope and others to the left child, then they write either 0 or 1, respectively, into the shared-memory array `left`. A reduction of the array `left` with the operation + allows to know if any ray has taken the left child direction. So, if the reduction is greater than 1, `NP` is set to its left child, otherwise it is set to rope.

9.3.2. Methodology for Testing the Roped BVH

We have implemented on CUDA a ray tracing of primary rays to test the performance of our SAH-built roped BVH. The GPUs we have used are a NVidia GeForce 8800 GTS (cap. 1.0) and a NVidia GeForce 280 GTX (cap. 1.3). The application is implemented with three kernels: *RayGenerator* (RG), *TraversalIntersection* (TI) and *Shading* (SH). The ray-triangle intersection algorithm we have used is due to Möller and Trumbore [MT97] and the ray-box intersection test is due to Shirley and Morley [SM03].

The traversal algorithm implemented in TI is a ray-packet roped-BVH traversal. In this algorithm, each ray is bound to a thread. Shared memory is used to store the information common to every ray in the packet, so there are only two choices to implement a packet: a packet is a thread block or a warp.

If the packet is implemented as a thread block, then the algorithm is called `packet-block`. Because of hardware requirements, blocks can own 256 rays/threads at most in GeForce 8800 GTS and 512 in GeForce 280 GTX. Specifically, we have tested 16, 32, 64, 128, 256 and 512 rays per packet. If the packet is implemented as a warp, it is called `packet-warp`. Because the size of a warp is 32 threads, there are always 32 rays per packet. Unlike `packet-block`, it is not necessary

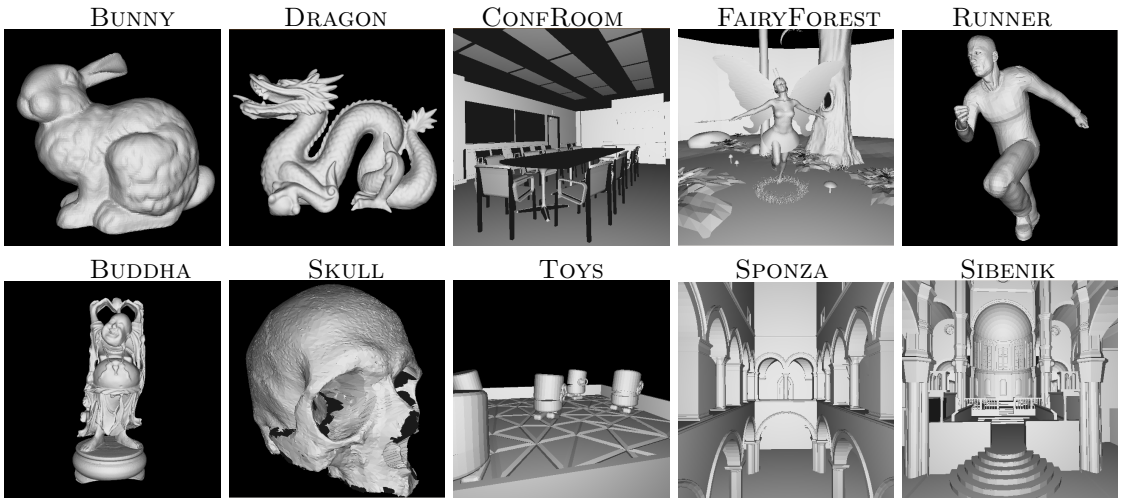


Figure 9.2: Snapshots of the scenes used to test our ray-packet roped-BVH traversal.

to use explicit barrier for synchronization.

We have tested our application on the scenes in Figure 9.2. The scenes Bunny, Dragon, Buddha, Skull and Runner have a single object composed of a great number of triangles. Toys is a very simple scene composed of few triangles. The scenes ConfRoom, FairyForest, Sponza and Sibenik have closed environments. All the images have resolutions of 512×512 and 1024×1024.

9.3.3. Results

The results of the applications for the resolution 1024×1024 can be retrieved in Table 9.1. The results for the resolution 512×512 are in Torres et al. [TMG09a]. The performance of `packet-warp` is always better than `packet-block`. The reasons why this happens are twofold. On one hand, `packet-warp` does not use explicit synchronization, so it has not the overload due to this instruction. On the other hand, the size of blocks are not bound to the size of the packet, so the optimal block size can be chosen.

9.4. Traversing a BVH Cut to Exploit Ray Coherence

Although the origins and directions of a set of rays are very different, they can be coherent if all of them intersect the same leaf node. This is due to the fact that at least there exists a pathway common to this set of rays during traversal. This pathway is the set of nodes from the root of the AS to the common leaf node.

One way to reach this coherence is to remove those rays that do not intersect that leaf during traversal. This can be achieved by filtering the rays that do not intersect some ancestor node of that leaf. So, we introduce the concept of *Cut* of a acceleration structure in our paper Torres et al. [TMG11]. A Cut comprises a set of subtrees of the AS that satisfies that each leaf of the AS only belongs to one of these subtrees (Figure 9.3). Because BVHs explicitly store the box of each node, we have implemented Cuts on BVHs.

Scene	Triangles	Geforce 280 GTX				Geforce 8800 GTS			
		`packet-warp`		`packet-block`		`packet-warp`		`packet-block`	
		FPS	shape	FPS	shape	FPS	shape	FPS	shape
Bunny	69,451	37.28	4×8	20.32	8×8	13.43	4×8	7.06	2×32
Dragon	871,414	21.60	8×4	10.35	8×8	6.73	4×8	3.17	4×16
Runner	78,029	41.67	2×16	19.80	1×64	16.91	4×8	9.34	2×32
FairyForest	174,117	22.55	4×8	11.82	2×32	7.97	4×8	4.25	2×32
Buda	1,087,716	24.33	2×16	9.51	1×64	8.65	4×8	3.92	2×32
ConfRoom	190,947	34.52	4×8	22.15	2×32	11.99	16×2	7.60	4×16
Sponza	66,454	26.49	8×4	15.97	8×8	8.55	4×8	4.95	2×32
Toys	11,141	50.72	4×8	33.23	8×8	19.80	4×8	12.14	4×16
Skull	102,905	29.64	4×8	14.32	2×32	10.32	4×8	5.15	2×32
Sibenik	80,479	23.74	4×8	14.79	8×8	7.56	4×8	4.51	4×16

Table 9.1: Best results for our ray tracing application on the GeForce 8800 GTS and the GeForce 280 GTX at a resolution of 1024×1024. The "FPS"-tagged columns are showing the *frames-per-second* of our best results of each scene. The "shape"-tagged columns are showing the shape of the packet that provides the results in the "FPS"-tagged columns.

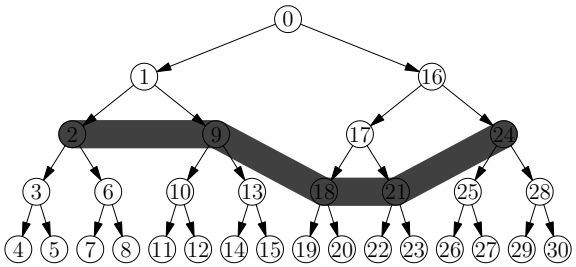


Figure 9.3: Example of a Cut of a BVH. The Cut comprises the subtrees hanging from the nodes 2, 9, 18, 21 and 24.

9.4.1. Cut Traversal

The traversal of a Cut is performed by separately traversing its subtrees. First, each ray tests the intersection with the box of the root of each subtree. The rays that do not intersect the box are ruled out, whereas the remaining ones traverse the subtree by using any GPU traversal algorithm. This operation is called *filter*. In our paper, we have tested two traversal algorithms: a packet-based (*persistent packet*) and a single-ray one (*persistent while-while*), both of them due to Aila and Laine [AL09].

The benefit of Cut traversal is twofold. On one hand, the traversal begins at nodes below the BVH root, which avoids the traversal of many nodes. On the other hand, the memory is exploited more efficiently. This is due to the fact that the BVH is stored in memory in depth-first order. So, the traversal of a BVH with a Cut entails that the used range of memory address is tighter and increases the probability of coalesced memory accesses and cache hits.

The benefit obtained by the use of Cuts is greater when roots of the Cut are deeper. However, the deeper the roots are, the greater the overhead due to filtering is. So, it is necessary to experimentally test the use of Cuts.

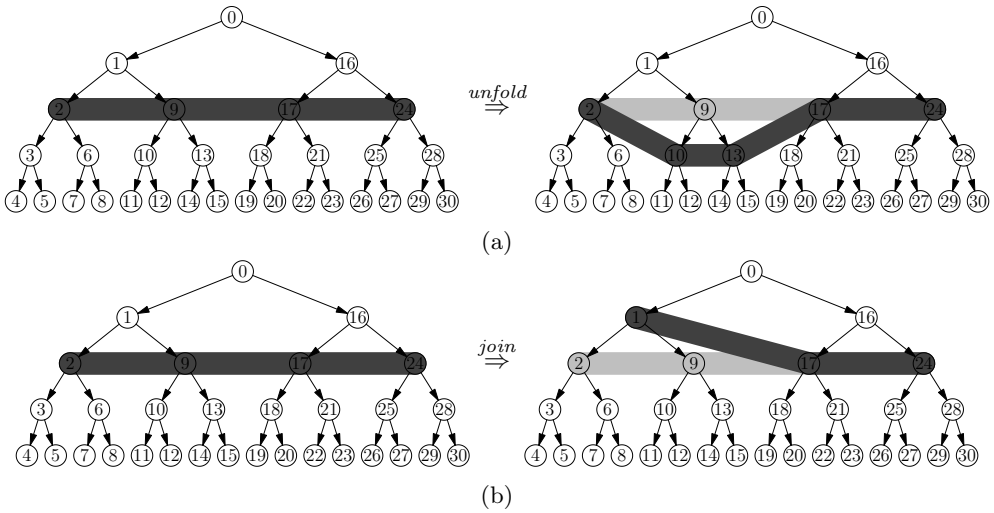


Figure 9.4: Fig. (a). Example of the operation *unfold*: the node 9 has been replaced with its two children: the nodes 10 and 13. Fig (b). Example of the operation *join*: the nodes 2 and 9 have been replaced with their parent: the node 1.

9.4.2. Cut Construction

We have used two techniques to construct Cuts: *structural construction* and *simulated annealing*. In structural construction, Cut's roots are selected by taking into account only properties of the nodes within the acceleration structure. In simulated annealing, the Cut is searched inside all the Cuts to find the minimum traversal-time Cut.

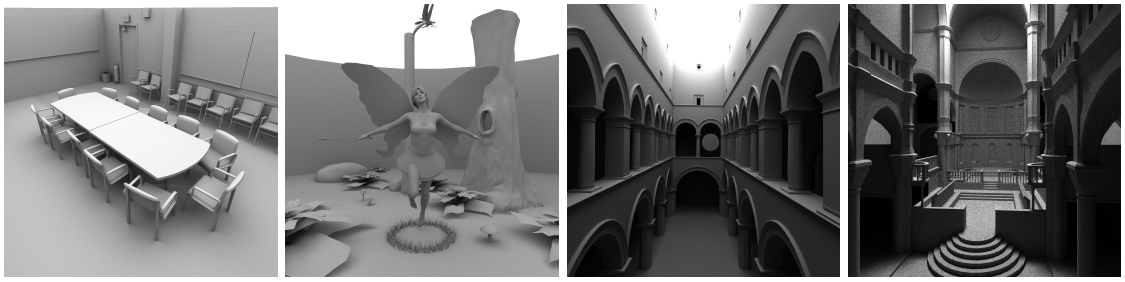
Structural Construction

In our paper, we have used depth and surface area of nodes as parameters during the Cut construction. In *depth structural construction*, the Cut comprises those nodes whose depth is a threshold. In *area structural construction*, the Cut comprises those nodes whose surface area is lesser than a threshold. In order to avoid some leaf to be missed, the leaves the construction algorithm reaches are directly added to the Cut, even if they do not conform the construction criterion.

Simulated annealing

The number of possible Cuts in a BVH is enormous, so an exhaustive search to find the minimum traversal-time Cut is not feasible. We have adapted the simulated annealing algorithm to find a Cut with good performance. The searching space is a connected, undirected graph of states, where each state is associated with an energy value. The simulated annealing algorithm tries to find the minimum state by probabilistically jumping between adjacent states according to their energies and to the global temperature.

We have adapted our problem to the simulated annealing scheme as follows. Each state of the graph is a Cut. The energy of the states is the total sum of the required time for a set of rays to be filtered and to traverse the subtrees of the Cut. Two states are mutually reachable if one of them is obtained from the other by using the operations *unfold* and *join* (Figure 9.4).



CONFROOM FAIRYFOREST SPONZA SIBENIK

Figure 9.5: Snapshots of the scenes used in our experiments.

9.4.3. Methodology

We have tested the performance of a BVH traversed with a Cut on a NVIDIA GeForce GTX 285 (cap. 1.3) with 1GB of RAM. The scenes used in our tests are FAIRYFOREST, CONFROOM, SPONZA and SIBENIK (Figure 9.5). All the images have a resolution of 1024×1024. The performance of the BVHs is increased by using the technique *early split clipping* due to Ernst and Greiner [EG07].

We have used a Path Tracing (PT) without Russian roulette as the ray tracing algorithm. All the materials in our scenes are diffuse. We call *generation* the set of rays that have the same number of bounces from the camera. Thus, rays in generation 0 are the primary rays, rays in generation 1 are the secondary rays, and so on. The coherence of rays gets worse with each bounce, reaching a very bad performance from generation 2 onwards. The number of rays in each generation is the maximum number allowed by our implementation and our GPU: 8MRays ($= 8 \cdot 2^{20}$ rays). Due to the fact that images have a resolution of 1024×1024, each 2×4 submatrix contains 8 samples for the same pixel. By using the Morton code, the elements of these 2×4 submatrices are adjacent on memory.

Our Path Tracing is implemented by five CUDA kernels: *RayGenerator* (*RG*), *Test*, *Compact*, *Traversal-Intersection* (*TI*) and *Shading* (*SH*). First, the primary rays are spawn from the camera in *RG*. After that, *Test* performs the ray-box intersection tests with the roots of the Cut. Non-intersecting rays are filtered in *Compact*. The kernel *TI* finds the nearest intersection point for each ray. The traversal algorithms we have used are *persistent packet* and *persistent while-while*. The kernel *SH* generates the next rays from the nearest intersection points.

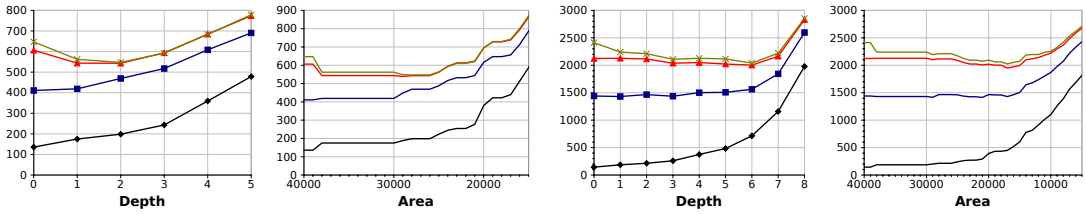
9.4.4. Results

Structural construction

We have tested the performance of several structural Cuts over different depth and area thresholds. In this section, we only show the results of the scene SPONZA. The remaining results can be retrieved in our paper Torres et al. [TMG11]. The rendering times of SPONZA scene are depicted in Figure 9.6. The x axis shows the thresholds used in the Cut construction. The y axis shows the time for traversal, measured in milliseconds (ms). A solid line has been drawn over the points corresponding to rays of the same generation. Although the 10 generations have been tested, only generations from 0 to 3 are shown. The remaining ones are very similar to generation 3 and they have not been shown for the sake of clarity.

The first value in the charts corresponds to the Cut whose subtree is the whole BVH. We call it Cut_{root}. Thus, the first value of each curve is the traversal time of the BVH plus the extra time of filtering. This time is around 10 ms according to our experiments.

The curves of each generation have a similar shape throughout the scenes. The generation 0



(a) Persistent while-while (b) Persistent packet

Figure 9.6: Rendering times (in *ms*) for SPONZA scene by using structural Cuts. The colours are: black (generation 0), blue (generation 1), red (generation 2) and green (generation 3).

and 1 do not undergo any improvement with respect to Cut_{root} (they are increasing curves). On the contrary, a valley appears at the beginning of the curves of generation 2 to 9 and later they undergo an increase again.

The improvement of each generation with respect to Cut_{root} is more important in *persistent packet* than *persistent while-while*. Because the overhead due to filtering is the same in both algorithms, we think hardware is responsible for this difference. If groups of rays are more coherent in *persistent while-while*, the number of read-from-memory nodes are the same, but the number of coalesced readings will increase. If ray packets in *persistent packet* are more coherent, the number of read nodes will decrease, but all readings are coalesced. The GPU memory system has greater benefit with the decrease of read nodes than with the increase of coalesced readings.

According to our results, the best saving percentages are in the scene SIBENIK (30.6% for depth, and 32.0% for area) for *persistent while-while*, and in the scene FAIRYFOREST (22.7% for depth and 40.9% for area) for *persistent packet*.

Simulated annealing

In general, simulated annealing finds Cuts with better performance that cannot be found in the structural construction. The best result for *persistent packet* is obtained in scene FAIRYFOREST, with 51.7% of saving. The average depth of the Cut is 5.9 and the average area is 17.9% out of surface area of the BVH root. For *persistent while-while*, the scene SIBENIK obtains the best result with 32.0% of saving for a Cut with 3.1 of average depth and 43.8% average area.

Chapter 10

Coherent-Path Generation

10.1. Introduction

In order to harness the coherence of a set of rays, two approaches can be used. The first one consists of randomly generating a set of rays and clustering them afterwards. The second approach is to directly generate coherent rays. This coherent-ray generation (or CRG) has the advantage that later clustering is not necessary. Because this generation is previous to traversal, heuristics criteria have to be used.

10.2. Path Generation

Supposing a ray has already found its nearest intersection point p. That ray has to choose the direction of the next ray of the path. That direction is obtained by randomly choosing a point ω on the surface of the unit hemisphere $\mathcal{H}_p$, placed on the intersection point p. The pdf of ω is determined by the BRDF of the object in the point p. So, first of all, a point $\omega^{(L)}$ is chosen on the canonical hemisphere $\mathcal{H}$ (local coordinates) with its pdf. Second, that point $\omega^{(L)}$ is transformed to the point ω in world coordinates.

By convention, the canonical hemisphere $\mathcal{H}$ is defined on the canonical base $\{(1,0,0)$, $(0,1,0)$, $(0,0,1)\}$, so that the vector $(0,0,1)$ points to its pole (Figure 10.1a). The hemisphere $\mathcal{H}_p$ is in world coordinates and it is placed on the intersection point p, so that the normal vector N_p points to the pole (Figure 10.1b).

In order to transform local coordinates of any point into world coordinates, it is necessary to find a *tangent space* on the intersection point of the ray. A tangent space consists of three orthonormal vectors $[U_p, V_p, N_p]$, so that N_p is the normal of the intersection point. Even though there are infinite tangent spaces, we will only use ones built with the next procedure. Let up be a vector non-parallel to the normal N_p. From these two vectors the tangent space is defined as

$$U_p = \frac{up \times N_p}{||up \times N_p||} \qquad \text{and} \qquad V_p = \frac{N_p \times U_p}{||N_p \times U_p||}$$

The vector ω in world coordinates can be obtained from the vector $\omega^{(L)} = (\omega_x^{(L)}, \omega_y^{(L)}, \omega_z^{(L)})$ in local coordinates and the previous tangent space as

$$\omega \quad = \quad \omega_x^{(L)} \cdot U_p \; + \; \omega_y^{(L)} \cdot V_p \; + \; \omega_z^{(L)} \cdot N_p$$

Suppose the vector up is common to every paths. If the normals on two intersection points are similar, then their tangent spaces, ET_1 and ET_2, will be also very similar. If the vector $\omega^{(L)}$

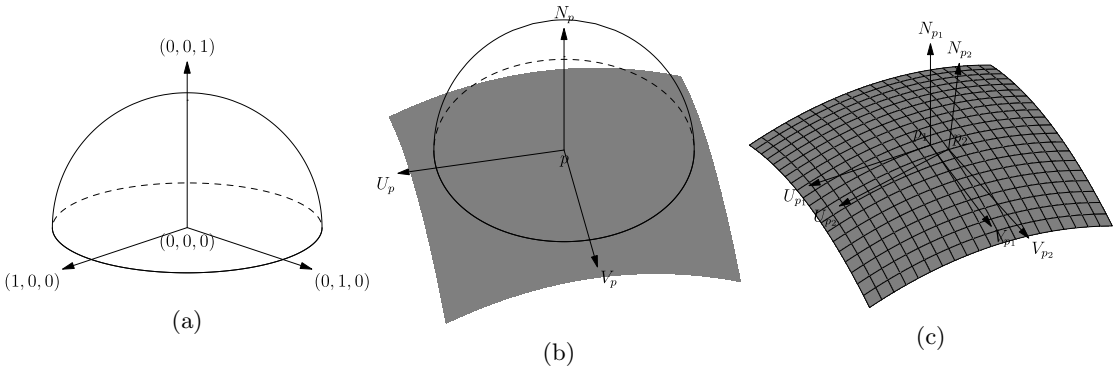


Figure 10.1: Fig. (a). The canonical hemisphere $\mathcal{H}$. The vector $(0, 0, 1)$ points to its pole. Fig (b). The hemisphere $\mathcal{H}_p$ placed on the point p. In this hemisphere, the normal vector of the surface N_p points its pole. Fig (c). Two tangent spaces placed on the same object and on close points. The same vector *up* has been used to build those tangent spaces.

is transformed to world coordinates by using the tangent spaces ET_1 and ET_2, two vectors ω_1 and ω_2 are obtained. Since these tangent spaces ET_1 and ET_2 are very similar, the vectors ω_1 and ω_2 will also be similar (Figure 10.1c). This fact is going to be used to generate coherent paths.

10.2.1. Generating Directions on a Spherical Patch

Let T be the function that transforms spherical coordinates into Cartesian coordinates

$$T(\theta, \phi) = \big(\sin(\theta)\cos(\phi),\ \sin(\theta)\sin(\phi),\ \cos(\theta)\big)$$

where $\theta \in [0, \frac{\pi}{2}]$ and $\phi \in [0, 2\pi]$. If the range of the coordinate θ is divided into Q parts and the range of the coordinate ϕ into M parts, the canonical hemisphere is divided into $Q \cdot M$ spherical patches (Figure 10.2). Each patch SP_{ij} is defined as

$$SP_{ij} = \Big\{ T(\theta, \phi) \quad | \quad \theta \in [i\Delta\theta, (i+1)\Delta\theta], \quad \phi \in [j\Delta\phi, (j+1)\Delta\phi] \Big\}$$

where $\Delta\theta = \frac{\pi/2}{Q}$, $\Delta\phi = \frac{2\pi}{M}$, $i \in \{0, 1, \ldots, Q-1\}$ and $j \in \{0, 1, \ldots, M-1\}$.

For a set of G rays, the generation of the new directions is as follows. First, just one spherical path is randomly chosen among the $Q \cdot M$ possible patches of the canonical hemisphere. After that, G points on that spherical patch are randomly selected. Finally, these points are transformed into world-coordinate directions with the tangent spaces on these intersection points.

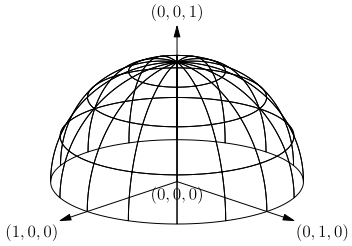


Figure 10.2: Spherical patches of the canonical hemisphere. The spherical coordinate θ is divided into 4 sections and the spherical coordinate ϕ into 16.

10.2.2. Coherent-Path Generation

Generating directions on the same spherical patch greatly ensures the sampling of coherent paths. So, suppose a set of G primary rays generated on the same pixel tile. These rays are geometrically very close, so they are very probable to be very coherent during their traversal and their nearest intersection points are also likely to be close. Next, the G rays choose new directions to extends their paths. If the vector *up* and the spherical path are common to all the G rays, all the generated directions will be very similar and the next rays will be likely to be coherent again.

Notice that there is not full guarantee for the rays to be coherent. E.g. two intersection points are possible to be close but their normal vectors could be different. So, although the vector *up* and the spherical patch are the same for both points, the tangent spaces are very different and the coherence will no be preserved.

10.3. Russian Roulette by Groups

During the rendering, a path can terminate before the others within a coherent group, which ruptures the coherence of the group. There are three reasons why a path terminates: it leaves the scene, it reaches a light, or it does not pass the Russian roulette test. The first two reasons of termination are due to the geometry of the scene, thus the coherent-path generation itself tends to avoid the rupture of a coherent group. The third reason, the Russian roulette, is the most probable cause of rupture because it is always difficult for all the paths of a coherent group to pass this test at the same time. So, we modify the Russian roulette test to keep the coherence of a ray group.

The Russian roulette test can rupture a coherent ray group mainly because this test is individually performed. A way to avoid this is that all rays of the group perform the test at the same time. We call this technique *Russian roulette by group*. So, the Russian roulette test is performed once and all rays of a group pass or fail the test at the same time.

10.4. Methodology

We have implemented the coherent-path generation and Russian roulette by group in our paper Torres et al. [TMG12]. We have used a Path Tracing (PT) implemented on CUDA as the rendering algorithm. This algorithm broadly follows the work by D. Antwerpen [Ant11]. The acceleration structure used is a SAH-built BVH.

The algorithm is implemented with four CUDA kernels: *Extends*, *Compact*, *Traversal* and *Display*. These kernels are sequentially executed in the aforementioned order. All the rendered images have a resolution of 1024×1024, and this is also the size of the set of rays during rendering.

The paths are stored on two arrays: `rays` and `idrays`. The array `rays` stores only the information needed to keep the state of paths, i.e. the origins and the directions of the last rays and the accumulated radiance. However, the arrangement of rays in this array do not follow the traversal order, because it is specified by the integer array `idrays`.

At the beginning of the rendering, the kernel *Extends* launches a thread per pixel. The identification number `id_g` of each thread represents the Morton code of its pixel. Each thread generates a primary ray through its pixel. This ray is saved in `rays[id_g]` and the index `id_g` is saved in `idrays[id_g]`. In addition, the threads in *Extends* initialize the pixels to black.

The kernel *Traversal* finds the nearest intersection point for each ray by following the order of `idrays`. This kernel is implemented by following the work by Aila and Laine [AL09]. Once the nearest intersection points for all rays are found, the kernel *Extends* extends the paths by generating directions for next rays. If coherent-path generation is not used, next directions are randomly chosen all over the hemisphere. *Extends* is also devoted to regenerate the paths from

Figure 10.3: Reference images used in our experiments. They have been redenred with `normal` in 20 minutes.

the camera when they terminate, similarly to Novak et al. [NHD10].

After *Extends*, the kernel *Compact* compacts the regenerated rays at the end of the array and the extended ones at the beginning. So, the coherence of the primary rays can be exploited. As mentioned before, this rearrangement is not directly performed on the array `rays`, but on the index array `idrays`. This kernel is implemented by using *radixsort* of CUDPP 1.1.1 [HOS+10] with one bit to sort.

Finally, the kernel *Display* shows the partial image. Its implementations is done to ease the interaction and debugging.

10.4.1. Implementation of the Coherent-Path Generation

A group of rays consists of G sequential rays according to the order in `idrays`. The sizes of G we have tested are connected to the thread hierarchy of CUDA. Specifically, we have tested sizes of a warp ($G = 32$), of a CUDA block ($G \in \{256, 512, 1024\}$) and of the grid ($G = 2^{20}$).

When the size is a warp or a block, a thread is devoted to select the spherical patch common to the group and to write it into shared memory. After that, the remaining threads read that information and extends their paths by using that spherical path. The names of these programs are `warp`, `block_256`, `block_512` and `block_1024` for $G = 32$, $G = 256$, 512 and 1024, respectively. When G is the grid, every ray of the set belongs to the same group. In this case, the CPU is devoted to randomly select the spherical patch common to all the rays. The name of this program is `grid`.

10.4.2. Implementation of Russian Roulette by Groups

The Russian roulette by groups is performed in the kernel *Extends*. If this test is passed, all the rays in the same group extend their paths. Otherwise, all of them finish and are regenerated. The probability of passing this test is the same as the single-ray Russian roulette.

With the use of Russian roulette by groups, the number of consecutive coherent rays is probable not to change. So, the compactation of the kernel *Compact* is not necessary. In our paper [TMG12] it is shown the results of our experiments with the execution and the removal of this kernel. Next Section shows only the results without *Compact*. The algorithms with Russian roulette by groups and without the kernel *Compact* are suffixed `_rr(NO)`.

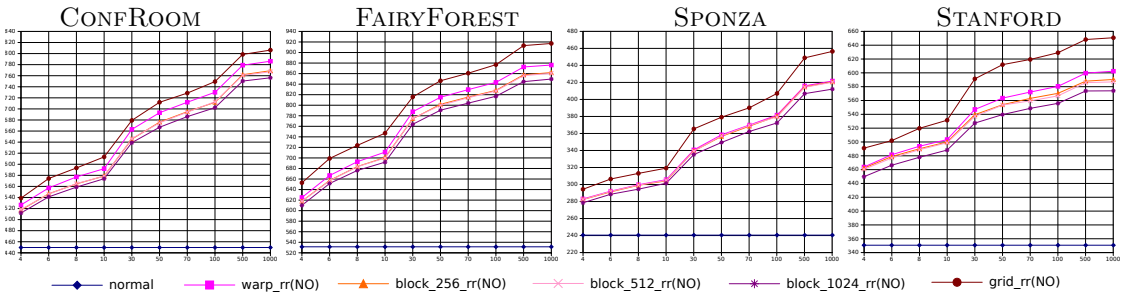


Figure 10.4: Number of terminated paths on average (axis y) with respect to Q, i.e. number of θ-coordinate divisions (axis x). The render time is always 30 seconds.

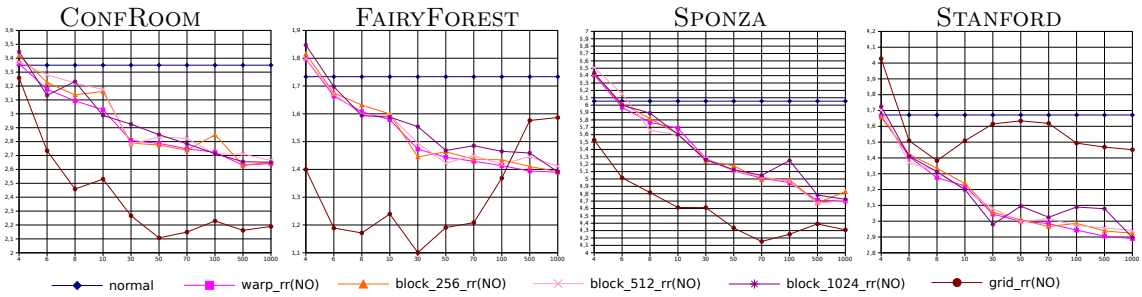


Figure 10.5: Root-Mean-Square Error (RMSE) with respect to the reference images. The axis x shows the number of divisions of θ, i.e. the parameter Q. The axis y shows the RMSE in percentage.

10.5. Results

We have tested our algorithms on the scenes ConfRoom, FairyForest, Sponza and Stanford (Figure 10.3). All the rendered images have a resolution of 1024×1024. We use 0.8 as the probability to pass the Russian roulette test.

The GPU used is a GeForce GTX 580 (cap. 2.0) with 1.5GB of RAM. The values of Q range from 4 to 1000, and, for the sake of simplicity, $M = 4Q$. The results have been compared to an ordinary Path Tracing, in which every path is extended individually. We denote this reference algorithm as `normal`. The reference images take 20 minutes to render with `normal` and the test images take 30 seconds of the kernel *Extends*, *Compact* and *Traversal* to render.

The charts in Figure 10.4 show the average number of terminated paths per pixel. The algorithm `normal` appears as a constant straight line because it does not deppend on Q. We reach the following conclusions. First, the number of terminated path for all the algorithms with Coherent-Path Generation is greater than `normal`. It is experimentally checked by *NVidia Visual Profiler* that the kernel *Traversal* is much faster due to less cache-miss rate and the total number of global-memory transferences is lower. Second, the higher Q is, the more similar the ray directions per group are, which is due to the fact that spherical patches are smaller. As a consequence, the curve lines for our algorithms are non-decreasing, i.e. the traversal is faster as Q is greater.

As Q becomes greater, the number of terminated path is greater, so this entails the error of the images is lower. However, the correlation of paths is also higher, which entails a bigger error. Figures 10.5 shows the error (measured as *Root-Mean-Square Error*, or *RMSE*) of each image with respect to the reference images. When Q is small, the error is lower than `normal` although the number of terminated paths is higher. As Q increase, the number of terminated path increases

and the error decreases until it becomes lower than the error of `normal`.

The correlation of the paths that appears when the Coherent-Path Generation algorithms is used makes a unpleasant visual pattern in the final images. In our tests, this pattern has been diminished by applying the technique called *interleaved sampling* [KH01].

Conclusions

Ray Tracing on GPU

The ray tracing algorithms can render photo-realistic images. The drawback is that it is necessary to trace a big amount of rays to obtain great-quality images. A naive GPU implementation of these algorithms seems straightforward, although it does not take the most of the hardware deeply.

The GPU memory is not able to provide data to functional units with enough bandwidth to keep them busy. The techniques used to reduce this inconvenient are to save the read-from-memory data into the fast on-chip memories and to reuse them in subsequent computation. In the ray tracing algorithms, the data read from memory are acceleration structure nodes and triangles of the scene.

If a set of rays is coherent, their rays consult the same nodes during the traversal, and, therefore, the memory accesses decrease and the cache-hit rate increases, so the whole performance rises. In this thesis, we have developed three algorithms to improve the GPU memory system. The first one is the development and implementation of a traversal algorithm on a roped BVH. This traversal makes it easier to use the strict memory coalesced-accessing pattern in GPUs of cap 1.0. The second algorithm we have proposed is the traversal of a BVH with Cut. It allows to increase the coherence of rays generated after several bounces. In the third algorithm, we have proposed to modify the path generation and termination. This brings the generation of geometrically coherent rays and the preservation of this coherence after each bounce.

Acceleration Structures

The acceleration structure is a factor to consider in the performance of ray tracing algorithms. The *top-down* algorithm along with the *Surface Area Heuristics* is the present standard procedure to build hierarchical acceleration structures. Many variations of SAH have been developed by changing some assumptions about rays or their traversal. These new heuristics overcome the performance of the classical SAH. However, it is still pending to prove the compatibility of all these methods and to compare the performance among them.

In Section 8.5, we have proposed variations of SAH oriented to rays whose directions are shrunk to a certain set. The traversal through the acceleration structures built with our heuristics leads to better whole performance. The drawback is the memory footprint due to the fact that three structures have to be stored in memory. This extra storage has to be taken into account by the application designer.

Parte III

Artículos Asociados con la Tesis

(Su orden de aparición coincide con el orden en que son citados en el resumen extendido de esta tesis)

Algorithmic Strategies for Optimizing the Parallel Reduction Primitive in CUDA

Pedro J. Martín, Luis F. Ayuso, Roberto Torres, Antonio Gavilanes
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, Spain
{pjmartin@sip, lf.ayuso@fdi, r.torres@fdi, agav@sip}.ucm.es

Abstract—**Many general-purpose applications exploit Graphics Processing Units (GPUs) by executing a set of well-known data-parallel primitives. Those primitives are usually invoked from the host many times, so their throughput has a great impact on the performance of the overall system. Thus, the study of novel algorithmic strategies to optimize their implementation on current devices is an interesting topic to the GPU community. In this paper we focus on optimizing the reduction primitive, which merely reduces a data sequence into a single value using a binary associative operator. Although tree-based and sequential-based algorithms have been already implemented on GPUs, a comparison of both algorithm performance had not been carried out yet. Thus, our first contribution is to present an experimental study of state-of-the-art reduction algorithms on CUDA. Next we introduce two algorithmic optimizations that are integrated into the fastest solution (a sequential-based algorithm), improving its throughput even more. Finally, we replicate this methodology to the segmented version of the primitive, which applies when the input is composed of several independent segments. In this case, it is not clear which algorithm exhibits the best performance, since throughput deeply depends on the distribution of segments along the input. According to our results, tree-based algorithms run faster for small segments, while sequential methods are better for medium and large ones.**

Keywords- parallel reduction; segmented parallel reduction; data-parallel algorithms; GPGPU; CUDA.

I. INTRODUCTION

Nowadays Graphics Processing Units (GPUs) are used to accelerate a wide range of general-purpose applications by exploiting their high-performance many-core processors. GPUs usually contribute to the overall computation in two different ways: carrying out some specific tasks through user-designed kernels or executing some data-parallel primitives provided by a growing number of libraries (e.g. CUDPP, CLPP, GPULib, Thrust…). While programmers assume control of the throughput of their kernels, primitives are integrated as black boxes whose performance relies on the library. On the other hand, hardware improvements incorporate more functional units in each release, along with larger shared memories and sophisticated cache hierarchies. For those reasons, the study of algorithmic strategies to optimize the implementation of these primitives, and the adjustment to the features of the upcoming GPU architectures, are ever-interesting topics to the General-Purpose GPU community [8].

In this paper, we focus on optimizing the classic *reduction primitive*, paying attention to its two versions. Its *unsegmented* form takes a binary associative operator $\oplus$ (e.g. $+$, $\times$, min and max) and an array of N data $[a_0, a_1, ..., a_{N-1}]$ as inputs, and it returns as output one value ($a_0 \oplus a_1 \oplus ... \oplus a_{N-1}$). In its *segmented* version, the input array is divided into segments of consecutive data, and the output is the individual reduction of each segment. Thus, the output size is the number of segments included in the given input. Observe that the segmented primitive could be easily implemented in terms of the unsegmented primitive, by extracting each segment and reducing it, isolated from the global input, with the unsegmented primitive. Nevertheless, this should be done many times (one for each segment), and some of the segments may be too small to justify further kernel executions. In consequence, this approach would not take the most of GPUs. On the contrary, the segmented solutions we illustrate will simultaneously perform separate parallel reductions on the segments of the input. For this reason unsegmented and segmented reductions are introduced as independent primitives along the paper.

The two reduction versions are useful building blocks for solving a wide variety of problems on GPU. For example and using CUDA, the unsegmented version has been successfully applied to solve the Single-Source Shortest-Path problem [12] and to build the Minimum Spanning Tree [16], while the segmented version to construct kd-trees on GPU for ray tracing [17] and to accelerate sparse-matrix multiplication [3].

Concerning the properties the operator $\oplus$ must fulfill, only associativity is essential. Actually, the correctness of all the algorithms described along this paper deeply relies on it. Most of the presented algorithms also make use of identity; so, $1_\oplus$ will denote an identity element for $\oplus$ from now on. Although the operator could be commutative as well, as it happens to the examples above, we will not suppose it in this paper, i.e. they are considered to be "non-commutative". The advantage of using commutativity, along with associativity, is that any pair of elements could be reduced regardless of their location on the input. However, these pairs must belong to the same segment in segmented reduction, which makes it difficult to exploit commutativity in this case. Hence, the exploitation of commutativity seems to come into conflict with the segment arrangement.

N	Number of elements in the input array
D	Size of a data-block
N_D	Number of data-blocks inside the input ($= ceil(N/D)$)
B	Number of threads in a CUDA-block
G	Number of CUDA-blocks in a grid
R	#Elements a thread loads in tree-based reductions ($= D/B$)
W	Width of the matrix in sequential matrix-reductions
H	Height of the matrix in sequential matrix-reductions
$MBPM$	Maximum number of resident blocks per multiprocessor
P	Number of producer warps in the producer-consumer scheme
C	Number of consumer warps in the producer-consumer scheme
$nBanks$	Number of banks in the shared memory of the device

Glossary. Parameters used in the paper.

One of the main aims of this paper is to compare two kinds of reduction approaches: recursive (tree-based) versus sequential algorithms. In fact, our first contribution is to experimentally confront the state-of-the-art algorithms of both types. Obviously, we have replicated the experimental study for the two versions of the primitive. As a second contribution, we propose two algorithmic optimizations that can be integrated into any of the previous algorithms, regardless their nature. We have tested them into the fastest solution, resulting in significant speed-up for unsegmented reduction. However, they did not lead to succeed in the segmented case since the resulting solutions are bounded by the shared memory size.

II. Related Work

The kernels presented by Harris [10] are the most popular CUDA implementations for the unsegmented reduction primitive. They are actually included as project examples in every CUDA SDK release. His document introduces seven kernels from a didactic perspective, in such a way that each kernel improves the performance of the previous one. Nevertheless, many of them require the operator to be commutative.

The two segmented reduction algorithms we have found are located at the previous references [17, 3], where the primitive is also applied to solve a specific problem. In both cases, those proposals are based on the works of Blelloch for the scan primitive below mentioned.

The reduction primitive is actually a part of the (*inclusive*) *scan* primitive, which has been studied more widely because of its great applicability [5]. Thus, we must describe in this section some of the progress made on the scan primitive. Scan also takes an array $[a_0, a_1, ..., a_{N-1}]$ and a binary associative operator $\oplus$ as inputs, but it returns an array containing the reduction of all the prefixes $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{N-1})]$. Scan also accepts a *segmented* version. In this case, the output is the unsegmented scan of each segment.

The classic parallel algorithms for scan were studied by Blelloch [4, 5]. His formulations were based on recursive equations whose application described a full binary tree. For this reason, they are called *tree-based* algorithms. Later on, with the advent of GPUs, these formulations were adapted to the novel programming model. Thus, tabulation techniques were applied to replace the recursive nature of tree-based algorithms with an iterative processing. Horn [11] was the first developing GPU-based implementations of the scan primitive. The complexity $O(N\log N)$ of its formulation was improved by Gress et al. [9] and Sengupta et al. [13] to a linear algorithm. The latter presents a work-efficient step-efficient implementation on CUDA that was adapted to the segmented case a year later [14]. In order to improve performance of previous algorithm, the authors exploit shared memory usage. However, the implementations involve bank conflicts, and the kernels may not scale well with shared memory size.

Subsequently, Dotsenko et al. [7] presented work-efficient sequential algorithms for the unsegmented and segmented scan. They decompose the input into blocks that are arranged as matrices in shared memory. Each matrix is sequentially reduced by rows and partial results are stored in a smaller array, which is processed later on. The overhead of previous tree-based formulations concerning synchronization barriers is reduced since each thread reduces a row. Immediately, Sengupta et al. [15] improved their tree-based algorithms incorporating an intra-warp operation: each warp individually performs a scan over 32 elements. Next, one warp carries out another intra-warp execution over the previous results to generate the reduction of the whole block. The advantages of this operation are that many synchronization barriers become unnecessary. However that paper does not include a comparison with the work by Dotsenko et al, thus, the most recent and fastest implementations of both trends –the intra-warp scan [15] for the tree-based trend and the sequential scan [7] for the sequential family– have not been experimentally compared yet.

III. Unsegmented Reduction

Along the paper, we use the term block to denote two different concepts. A *CUDA-block* is a block of threads, while a *data-block* is a chunk of consecutive data that is individually reduced by a CUDA-block. As we will see later, a CUDA-block can reduce one or several data-blocks. We use B and D to denote the sizes of a CUDA-block and a data-block, respectively (see Glossary).

A data-block is loaded from global memory to shared memory by the corresponding CUDA-block, before being reduced. This is done exploiting coalesced readings. When the input size N is not a multiple of D, we append a virtual padding to the last data-block, which is filled with values $1_{\oplus}$. This is done through an *if*-statement during the loading stage. Thus, all the algorithms of this paper work on $N_D = ceil(N/D)$ data-blocks.

In order to reveal the main differences among the algorithms, we focus on how a data-block is reduced into a single result, and on how this value is handled afterwards. Thus, we will suppose that the array `s_data` holds a data-block in shared memory for subsequent reduction.

A. State-of-the-art Reductions

A common characteristic of these algorithms is that each CUDA-block exactly reduces a data-block. The result is then written back into global memory by a thread of the CUDA-block. Hence, the grid size is $G = N_D$, which also corresponds to the output size. In consequence, the output must be reduced again with another kernel launch, and so on, until a single result is left. This is usually known as a *multi-pass* approach.

1) Tree-based reductions: Supposing that `s_data` lays on the leaves of a full binary tree, tree-based algorithms reduce each pair of siblings using ⊕, in a bottom-up manner. Hence, they require D to be a power of two. The underlying recursive equations are:

$$\mathrm{red}(n) = \begin{cases} \mathrm{red}(left(n)) \oplus \mathrm{red}(right(n)) & \text{if } n \text{ is inner} \\ n & \text{if } n \text{ is leaf} \end{cases} \quad (1)$$

Algorithm 1 is similar to the kernel#2 by Harris [10]. At each iteration, a thread reduces two elements at line 14 (with indices `ai` and `bi`), and stores the result in `ai`, which corresponds to a *left-storing* approach. Storing into `bi` would be also possible (*right-storing*). The first iteration requires a thread for each pair of elements, thus $D = 2B$ and B must be a power of two as well. So the algorithm executes the loop $log_2(2B)$ times. Also observe that in each iteration, the number of active threads is halved (line 7), reaching a single active thread in the last iteration.

In order to avoid bank conflicts in shared memory, Harris replaces this interleaved addressing access with a sequential addressing pattern in his kernel#3. This new approach requires ⊕ to be commutative, so this technique has not been considered in this paper. On the contrary, we overcome bank conflicts by including the usual padding of one element every *nBanks* elements, where *nBanks* is the number of banks in the shared memory of the device. Thus, the total padding is *2B/nBanks* elements. This is why we update the indices at lines 11 and 12. Incorporating this offset does not penalize occupancy on current devices, and the overhead due to the index arithmetic is negligible.

Tree-based algorithms are quite fast, but they suffer from too much synchronization. Notice that a barrier must be located between iterations (line 6). However, the number R of elements that a thread loads from global memory can be tuned to achieve a certain speed-up. Notice that $D = R * B$ then holds. We do not consider such improvements as algorithmic optimizations, but code optimizations, so we have not paid attention to them in this paper. Observe that R=2 in Algorithm 1. On the contrary, R=8 in the scan implementation of CUDPP 1.1.1 [6].

```
1  void TB2_reduction (float* s_data){
2    unsigned int thid = threadIdx.x;
3    unsigned int stride = 1;
4    unsigned int i, ai, bi;
5    for(unsigned int d=B; d>0; d>>=1){
6      __syncthreads();
7      if(thid < d){
8        i = 2*stride*thid;
9        ai = i;
10       bi = ai + stride;
11       ai += (ai >> 5); //log2(nBanks)=5
12       bi += (bi >> 5); //log2(nBanks)=5
13       //Reduction
14       s_data[ai] = op(s_data[ai], s_data[bi]);
15     }//if
16     stride <<=1;
17   }//for
18   //The result is in s_data[0]
19 }
```

Algorithm 1. Tree-based reduction. R=2.

```
1  void tb_reduceWarp( float* s_data,
2                      unsigned int thid,
3                      unsigned int lane,
4                      float& target){
5   //REDUCTION INTRA-WARP (Left-Storing)
6   if( !(lane & 1)  ) s_data[thid]=
7     op(s_data[thid], s_data[thid+1]); //%2=0
8   if( !(lane & 3)  ) s_data[thid]=
9     op(s_data[thid], s_data[thid+2]); //%4=0
10  if( !(lane & 7)  ) s_data[thid]=
11    op(s_data[thid], s_data[thid+4]); //%8=0
12  if( !(lane & 15) ) s_data[thid]=
13    op(s_data[thid], s_data[thid+8]); //%16=0
14  if( !(lane & 31) ) target=
15    op(s_data[thid], s_data[thid+16]);//%32=0
16 }
17 // ************************************
18 //D=1024, B=256 => Nwarps=8, 32 intermediate results
19 void TB4_Warp_reduction (float* s_data){
20  __shared__ float s_result[32];//32 results
21  unsigned int thid   = threadIdx.x;
22  unsigned int warpid = thid >> 5; //thid/32
23  unsigned int lane   = thid & 31; //thid%32
24
25  //Reduce s_data[kB+warpid*32, kB+(warpid+1)*32)
26  //and store the result into s_result[k*Nwarps+warpid]
27  for(unsigned int k = 0; k<4; k++)
28    tb_reduceWarp(s_data, thid+k*blockDim.x, lane,
29                  s_result[warpid+k*Nwarps]);
30  __syncthreads();
31
32  if(warpid==0)
33    tb_reduceWarp(s_data, thid, lane, s_data[0]);
34  //The final result is in s_data[0]
35 }
```

Algorithm 2. Intra-warp tree-based reduction. B=2^8, D=2^{10}.

2) Intra-warp tree-based reductions: Sengupta et al. [15] improve the previous tree-based algorithm by working at the warp level. In order to explain this techique, let us extend the notation of CUDA-block and data-block to the case of warps. Briefly, a *CUDA-warp* is composed of 32 adjacent threads inside a CUDA-block, which run implicitely synchronized in SIMD fashion, while a *data-warp* is a chunk of 32 consecutive data inside a data-block. Sengupta et al. avoid bank conflicts and many synchronization barriers, using an intra-warp routine in which each data-warp is scanned by a CUDA-warp. The device function `tb_reduceWarp` of Algorithm 2 adapts this tecnique to the reduction primitive. During its execution, each CUDA-warp is responsible for reducing one data-warp, and sending the result to parameter `target`. The five iterations that are enough to reduce a data-warp have been unrolled, as the authors do. Notice that no explicit synchronization barriers are requiered due to the way CUDA-warps run in CUDA.

The kernel `TB4_Warp_reduction` of Algorithm 2 repeatedly invokes the previous function to reduce the whole data-block. It uses a shared array called `s_result` to hold the intermediate result each data-warp produces. We have used D=1024 and B=256, thus R=4, there are N_{warps}=8 CUDA-warps, which are responsible for reducing 4 data-warps, and the number of intermediate results is 32. Each data-warp is reduced at line 28, and the result is sent to `s_result[warpid+k*`N_{warps}`]`. Finally, intermediate results are reduced again at line 33 by the first CUDA-warp. The final result is sent to `s_data[0]` for the sake of clarity; in practice it is sent straight to global memory.

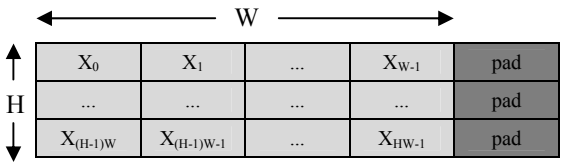


Figure 1. Arranging `s_data` as a matrix.

3) Sequential reductions: Dotsenko et al. [7] propose an algorithm for the scan primitive that is based on a matrix representation of `s_data`, as Fig. 1 shows. H and W respectively denote the height and width of this matrix, so $D = H \times W$. Algorithm 3 adapts their *MatrixScan* to the reduction primitive. Observe that one thread is responsible for sequentially reducing one row of W elements, thus only H threads are required to reduce the whole data-block (line 5). Nevertheless, all the threads inside the CUDA-block cooperated at the beginning to load the data, including these H threads.

Since reducing a row involves no synchronizations, the performance of the algorithm could be improved by maximizing W. Nevertheless, D must be small enough to fit in shared memory, thus W and H are forced to be rather small as well. In addition, H should be a multiple of the CUDA-warp size in order to avoid divergent warps. To sum up, Dotsenko et al. finally assign W and H to be the warp size ($W = H = 32$). A padding of one element is then added at the end of each row to avoid bank conflicts. Moreover, the H values that are obtained after reducing the H rows can be reduced using one intra-warp tree-based reduction (line 16). Also notice that only one warp continues beyond line 5 since $H = 32$, so no explicit synchronization is needed at line 15.

B. Algorithmic Optimizations

We present two algorithmic optimizations that can be integrated into any of the solutions described above.

1) Persistent blocks: We can force each CUDA-block to reduce multiple consecutive data-blocks, instead of a single one. Thus, the output size decreases since the number of CUDA-blocks (G) is smaller than the number of data-blocks. Moreover, the number of multipasses falls as G decreases. The

```
1   void Matrix_reduction (float* s_data){
2    unsigned int thid = threadIdx.x;
3    float current_red;
4    //Only the first threads reduce
5    if(thid < H){
6     unsigned int s_base = thid * (W + PADDING);
7     float* row = &s_data[s_base];
8     current_red = row[0];
9
10    for(unsigned int k=1; k<W; k++)
11       current_red = op(current_red, row[k]);
12
13    //Store the reduction of the row
14    s_data[thid] = current_red;
15
16    tb_reduceWarp(s_data, thid, thid&31, s_data[0]);
17   }//if
18  } //The final result is in s_data[0]
```

Algorithm 3. Sequential matrix-based reduction. H=W=32.

reductions of the data-blocks assigned to a CUDA-block are accumulated using a shared variable. Specifically, one of its threads accumulates the result of a data-block into the reduction of the previous data-blocks.

In order to distribute all the data-blocks among all the CUDA-blocks, we simply incorporate two new parameters into the kernel to indicate the quotient `q` and the remainder `r` of the division N_D/G. Then, a CUDA-block must reduce `q+1` data-blocks if `blockIdx.x<r`, or just `q` data-blocks otherwise.

In order to improve performance, the grid size must be carefully chosen according to the requirements of the kernel. Given a block size B, the maximum number of resident CUDA-blocks per multiprocessor (*MBPM*) is computed according to the CUDA Occupancy Calculator. Then G is fixed to *NUM_MULTIPROCESSORS*MBPM*. The underlying idea is to fill each multiprocessor with the maximum number of CUDA-blocks that can reside together on it. Thus, no CUDA-warp will wait for being allocated on a multiprocessor. We use the adjective *persistent* to denote these CUDA-blocks since they will be residing on the device until the whole input is processed.

Using persistent blocks results in a small grid size, since the number of multiprocessors is limited by the device, and *MBPM* cannot exceed 8 in any of the current CUDA compute capabilities. Thus, one kernel execution is almost enough to reduce the whole input. Indeed, the results after the first launch do not even complete a data-block because $G < D$. In consequence, we remove the usual recursion controlled by the host, and furthermore the cost of storing partial results into global memory between recursive calls.

The idea of persistent blocks has been already applied to other topics. For example, it was recently used to accelerate the traversal step of GPU-implemented ray tracers by Aila and Lane [1], under the term *persistent thread*. In fact, Harris [10] already proposed a reduction algorithm, the so-called *cascading* kernel#7, whose CUDA-blocks reduce many data-blocks, but during the load stage, rather than during the reduction stage, and using a commutative operator. Nevertheless, these authors do not explain how the grid size (G) is chosen in their respective papers.

2) Producer-consumer scheme: We add another optimization onto the persistent technique in order to help the schedulers to hide memory latency a little more. The idea is to classify the set of CUDA-warps into two groups: consumers and producers, of respective sizes C and P. At each iteration, consumer warps sequentially reduce the data-block loaded in the previous iteration, while producer warps load a new data-block. Thus, consumers can reduce at the same time producers load new data.

Algorithm 4 shows the code fragment that implements the producer-consumer scheme. Two data-blocks are held in shared memory, which are accessed through two pointers, `s_load` and `s_comp`. These pointers are swapped at the beginning of each iteration (line 10). Then, consumer warps reduce the data-block pointed by `s_comp` (line 13), while producer warps load the

```
1  //Load the first data-block
2  if(warpid>=C)
3    loadChunk(first, s_load);
4  __syncthreads();
5  first += D;
6
7  //Producer-consumer loop
8  for(unsigned int k=1; k<d; k++){
9    //swap shared buffer pointers
10   aux = s_load; s_load = s_comp; s_comp = aux;
11   //Each thread does its job
12   if(warpid<C)
13      reduceChunk(s_comp, s_current_red);
14   else
15      loadChunk(first, s_load);
16   __syncthreads();
17   first += D;
18 } //for
19
20 //Reduce the last data-block
21 if(warpid<C)
22   reduceChunk(s_load, s_current_red);
23 __syncthreads();
24 //The result is in s_current_red
```

Algorithm 4. Producer-consumer scheme.

data-block that is located at index `first` in global memory into the buffer pointed by `s_load` (line 15). Observe that a synchronization barrier is required (line 16) to prevent warps from overwriting the other buffer. Also notice that the first/last data-block is loaded/reduced before/after the loop. The number of iterations is controlled by variable `d`, which holds the number of data-blocks assigned to this CUDA-block. The result of reducing a data-block is accumulated into the shared variable `s_current_red` inside the `reduceChunk` routine.

Fig. 2 graphically exposes the advantages of this technique, using the sequential matrix-based solution presented in Algorithm 3 as the underlying reduction method. Remember that each data-block is arranged as a matrix of size $D = H \times W = 1024$, where $H = 32$ and $W = 32$ denote its height and width, respectively. The figure depicts how warps can be dispatched if the optimization is incorporated (on the left), and if is not (on the right). In the first case, the producer-consumer scheme has a configuration of $P = 8$ producers (warps from W_1 to W_8) versus $C = 1$ consumers (warp W_0). Observe that a single consumer warp is enough, since exactly 32 rows must be reduced. In addition, each producer is responsible for loading

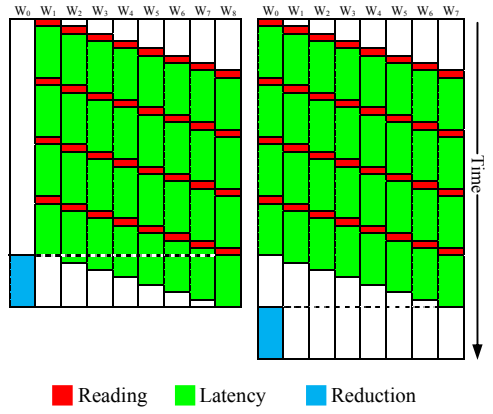


Figure 2. Producer-consumer scheme (left). Persistent sequential matrix-based reduction (right).

$D/P = 128$ elements, which is done in $128/32 = 4$ coalesced readings. Whenever a producer has requested data from global memory, it must wait until those data are available. This waiting time can be used to request more data by another producer, or to reduce the previous data-block by the consumer.

On the right, the figure shows the execution of the persistent sequential matrix-based algorithm without the optimization. In this case we have 8 warps in a CUDA-block ($B = 256$) processing $D = 1024$ elements as before. Notice that warp W_0 starts reducing only when all the data-block are available in shared memory. Thus, each iteration is slightly longer than one using the producer-consumer scheme.

Although the producer-consumer paradigm is a well-known technique, its implementation on GPU is a recent issue. Besides our paper, Bauer et al. apply DMA techniques to solve in GPU other problems [2]. Our scheme can be compared to their "manual double-buffering" technique.

IV. Segmented Reduction

In segmented reduction (*s-reduction* in the sequel), the input is divided into *segments*. We demarcate them by using another array of size N, called `owner`, such that `owner[i]` holds the index of the segment of element `i`. Hence, `owner` is sorted in non-decreasing order. Notice that the output of the s-reduction is an array whose size is the number of segments. In the sequel, the variable `g_output` will denote such array, which is located on global memory.

Next we adapt the algorithms presented so far to the segmented case. Again we focus on the reduction step, since it exposes the differences among them. Thus, we suppose that data and owners already hold in shared memory, specifically in `s_data` and `s_owner`.

A. State-of-the-art Segmented Reductions

Zhou et al. [17] propose a tree-based algorithm by adapting (1) to the segmented case. The underlying recurrences are:

$$\text{s-red}(n) = \begin{cases} \text{s-red}(l(n)) \oplus \text{s-red}(r(n)) & \begin{bmatrix} n \text{ is inner, and} \\ \text{owner}(l(n)) = \text{owner}(r(n)) \end{bmatrix} \\ \text{s-red}(l(n)) & \begin{bmatrix} n \text{ is inner, and} \\ \text{owner}(l(n)) \neq \text{owner}(r(n)) \end{bmatrix}^{\dagger} \\ n & [n \text{ is leaf} \end{cases}$$

$$\text{owner}(n) = \begin{cases} \text{owner}(l(n)) & n \text{ is inner} \\ \bar{n} & n \text{ is leaf} \end{cases}$$

Expressions $l(i)$ and $r(i)$ respectively denote the left and right children of an inner node i, and $\bar{n}$ is the owner of leaf n. Zhou et al. include an extra operation that must be applied when the siblings have different owners (label †). In this case, the s-reduction of the right child must be accumulated into the global solution. Specifically, each thread stores in `ai` the reduction of two elements, only if their owners are the same. Otherwise, only the left one is propagated, while the other is used to update `g_output`. Hence, the prefix of the processed chunk is propagated in a bottom-up manner, which agrees with this left-storing approach. Fig. 3 shows an example of a tree-

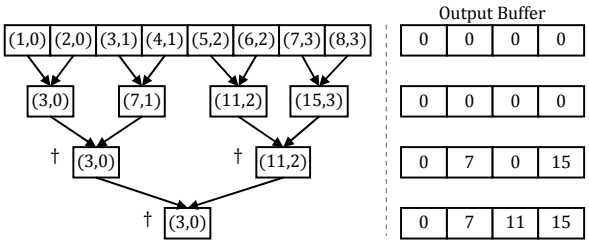


Figure 3. Example of a tree-based s-reduction. The reduction of the first segment has not been written to the output buffer yet.

based s-reduction for the + operator; on the left it illustrates the reduction process where each box contains a pair *(datum,owner)*, on the right it shows how the output buffer evolves as the algorithm progresses. Notice that the buffer is initialized with $1_{\oplus}$ values.

In order to adapt Algorithm 1 to the segmented case, it is enough to replace line 14 with the following code fragment:

```
14   unsigned int lo = s_owner[ai]; //left  owner
15   unsigned int ro = s_owner[bi]; //right owner
16   if(lo!=ro)
17      g_output[ro] = op(g_output[ro], s_data[bi]);
18   else
19      s_data[ai] = op(s_data[ai], s_data[bi]);
```

The partial result of the first segment inside this data-block ends at `s_data[0]` and `s_owner[0]`. The kernel is responsible for s-reducing a data-block, thus a multi-pass approach is required again to completely s-reduce a larger input. This is common to all the algorithms we present in this subsection.

Warps can also be exploited to improve Zhou et al.'s algorithm by avoiding some synchronizations. Basically, we must replace lines 7, 9, 11, 13 and 15 of Algorithm 2 with a proper call to the following device routine, in order to obtain `tb_s_reduceWarp`:

```
1    void s_reducePair_toTheLeft(float* g_output,
2           unsigned int oLeft,  float &dLeft,
3           unsigned int oRight, float dRight){
4      if(oLeft!=oRight)
5        g_output[oRight] = op(g_output[oRight], dRight);
6      else dLeft = op(dLeft, dRight);
7    }
```

Notice that prefixes are propagated again. Replacing in `TB4_Warp_reduction` any call to `tb_reduceWarp` with a call to `tb_s_reduceWarp` results in the intra-warp tree-based kernel for s-reduction.

Concerning sequential matrix-based reduction, we must replace lines 7-16 of Algorithm 3 with the following code fragment to cover the segmented case:

```
7    float* dRow = &s_data[s_base];
8    unsigned int* oRow = &s_owner[s_base];
9    //The first element is specially processed
10   current_red   = dRow[0];
11   current_owner = oRow[0];
```

```
12   for(unsigned int k=1; k<W; k++){
13     if(current_owner!=oRow[k]){
14       g_output[current_owner] =
15             op(current_red, g_output[current_owner]);
16       current_owner = oRow[k];
17       current_red   = dRow[k];
18     }else
19       current_red = op(current_red, dRow[k]);
20   }//for
21   //Store the reduction of the row
22   s_data [thid] = current_red;
23   s_owner[thid] = current_owner;
24   tb_s_reduceWarp_toTheRight(s_data, s_owner, g_output,
25                 thid, thid&31, s_data[0], s_owner[0]);
```

Data layout is left-to-right, top-to-bottom, hence the suffix of the processed chunk inside the row is propagated now (line 12), and the tree-based s-reduction at lines 24-25 must be right-storing. The partial result of the last segment is sent to `s_data[0]` and `s_owner[0]`.

B. *Algorithmic Optimizations*

The techniques we proposed for the unsegmented case can be integrated into the previous state-of-the-art segmented algorithms. Specifically, we incorporate persistent blocks, and the producer-consumer scheme afterward, into the sequential matrix-based algorithm. This latter technique requires a considerable amount of shared memory, since two data-blocks for the elements and another two data-blocks for the owners are needed at the same time for a CUDA-block. Thus, occupancy decreases on current devices, which turns into a not competitive performance as we will see.

V. EMPIRICAL RESULTS AND DISCUSSION

We have used a NVIDIA GTX 480 (capability 2.0 -Fermi, 480 cores, 1536MB of GDDR5 global memory, configured as 48KB of shared memory per multiprocessor and 16KB of L1 cache), with driver 270.61, and the CUDA Toolkit, SDK and Compute Visual Profiler 4.0.

This paper focuses on empirically comparing algorithms by testing their straight implementations. Thus, we have not tuned the code as much as possible. In the experiments described below, the operator is the *minimum* function on *float* data (4 bytes). The timing information was obtained by reducing ten times a random input, and by taking their performance on average. The input size is $N=m*2^{20}$, where m ranges from 16 to 31. We also tested other operators, e.g. + on *float* data, obtaining similar runtimes that are not included in the paper.

Concerning global memory accesses, Fermi architecture incorporates an on-chip cache hierarchy which is fairly configurable. Specifically, accesses can be cached in both L1 and L2, which is the default setting, or in L2 only. Since our implementations report similar runtimes under both configurations, the results we present below correspond to the default mode (L1 and L2).

A. *Unsegmented Reduction Results*

We have tested the five algorithms previously presented:

- `TB2:` the tree-based solution of Algorithm 1, with $R = 2$ and $D = 2B$.
- `TB4-warp:` the intra-warp tree-based solution included in Algorithm 2, with $R = 4$ and $D = 4B$.

- `Matrix`: the sequential solution of Algorithm 3, with $H = W = 32$, and $D = 1024$.
- `Persistent Matrix`: the previous `Matrix` solution incorporating persistent blocks.
- `Diffwarps`: the previous `Persistent Matrix` solution incorporating the producer-consumer scheme described in Algorithm 4. We use the term `Diffwarps` to express that warps carry out different tasks.

Notice that we have only incorporated the algorithmic optimizations into the sequential matrix-based reduction. The reason is that `Matrix` exhibits the best performance for unsegmented reduction.

The features of the five implementations are presented in Table I on the left. It includes *MBPM* for persistent solutions because it determines the corresponding grid size. Runtimes and effective bandwidths are shown in Fig. 4. `Diffwarps` exhibits the best throughput, with a bandwidth near to 104 GB/s. If B_r and B_w denote the number of bytes read and written by the algorithm, and $time$ is the runtime in seconds, effective bandwidth has been calculated as $((B_r + B_w)/10^9)/time$.

Table I on the right shows some empirical details for $30*2^{20}$ elements. These columns are especially relevant since they exhibit the behavior of each kernel execution individually. Notice that non-persistent solutions (`TB2`, `TB4_Warp` and `Matrix`) require three launches to completely reduce the input, while persistent ones (`P_Matrix` and `Diffwarps`) only need two. According to the Profiler reports, we include the following runtime information: grid size, global memory read throughput and average number of warps that are active on a multiprocessor per cycle *aw/ac*, which is calculated as (active warps)/(active cycles). Notice that bandwidth and *aw/ac* decrease as the reduction process advances. This is because the input for the first launch is larger than the input for consecutive launches. Hence, the GPU has less workload and becomes less efficient for subsequent launches, which explains why persistent solutions run faster.

B. Segmented Reduction Results

We have tested the corresponding five segmented algorithms. Their features are shown in Table II. Observe that the theoretical occupancy of most algorithms is smaller than those of their unsegmented counterparts. In the case of `Diffwarps`, it is so small (38%) that it is not competitive. Thus, we have added a variant which sequentially s-reduces a smaller matrix (H=32, W=16). Let $\texttt{Diffwarps}_{16}$ denote such solution. Notice that we then recover the 94% occupancy of the unsegmented version.

The way segments are distributed has a profound impact on the performance of all the algorithms, since the accesses to global memory, which are required when the left and right owners are different, called *irregular acce*sses in the sequel, are irregularly spread along execution. Thus, we have tested three scenarios: (a) a single huge segment covering the whole data, (b) segments of random size, ranging from 10 to 50, and (c) many small segments of size 3.

Fig. 5 shows the runtimes we have obtained for the three cases. Concerning state-of-the-art reductions, the figure shows two surprising facts: (1) `TB2` runs faster than `TB4-warp` in the three scenarios, and (2) tree-based solutions exhibit a better performance w.r.t. sequential ones as the segment size gets smaller. We will explain the reasons we find below. With regards to the algorithmic optimizations, `P_Matrix` and `Matrix` show similar performance in the three cases, and `Diffwarps` and $\texttt{Diffwarps}_{16}$ are not competitive in general, although the latter exhibits higher throughput.

Let us focus on scenario (a), that is, only one segment appears. Since irregular accesses do not take place, the results should be similar to those obtained for unsegmented reduction. Two facts remain: matrix-based beat tree-based methods, and `P_Matrix` improves `Matrix`; but two new issues come up. On the one hand, $\texttt{Diffwarps}_{16}$ is not among the fastest solutions in segmented reduction, while `Diffwarps` was the fastest in the unsegmented case. Each data-block requires loading the same amount of bytes in $\texttt{Diffwarps}_{16}$ (segmented) as in `Diffwarps` (unsegmented), since the matrix is halved (W=16) but owners are also loaded. On the contrary, the data size that is processed in a data-block is halved in $\texttt{Diffwarps}_{16}$ (W=16). Hence, read bandwidth gets halved, which finally results in a suboptimal performance. The Profiler endorses such claim since global memory read throughput falls from 104.17 GB/s in `Diffwarps` (unsegmented) to 53.67 GB/s in $\texttt{Diffwarps}_{16}$ (segmented), for the first launch on $30*2^{20}$ elements.

On the other hand, the relation between `TB2` and `TB4-warp` gets reversed from unsegmented to segmented reduction. `TB4-warp` has the advantage of requiring less explicit synchronization barriers, but it presents more intra-warp divergences because many threads inside a CUDA-warp are stalled inside function `tb_reduceWarp`. These divergences cause more harm to the segmented version of `TB4-warp` than to its unsegmented counterpart, because the divergent code is heavier in the segmented case due to `s_reducePair_toTheLeft`. Table III proves such assumption by showing the percentage of divergent branches the Profiler reports for $30*2^{20}$ elements. Observe that `TB2` and `TB4-warp` exhibit a similar divergence for unsegmented reduction, while `TB4-warp` is around ten times

TABLE I. KERNEL FEATURES FOR REDUCTION (LEFT). PROFILER'S REPORT FOR $30*2^{20}$ DATA (RIGHT).

Solution	#Registers	Shared Memory	Block Size	Theoretical Occupancy	MBPM	N= $30*2^{20}$		
						#Blocks	Global Memory Read Throughput (GB/s)	aw/ac
TB2	9	2B+2B/nBanks	256	100%	-	61440, 120, 1	25.53, 17.58, 0.11	46.24, 36.37, 7.88
TB4-warp	13	4B+32	256	100%	-	30720, 30, 1	36.36, 16.45, 0.03	45.12, 22.21, 7.35
Matrix	10	H×(W+1)	256	100%	-	30720, 30, 1	92.88, 26.33, 0.01	39.48, 20.85, 6.24
Persistent Matrix	15	H×(W+1)	256	100%	6	90, 1	94.25, 0.68	45.25, 7.82
DiffWarps	15	2(H×(W+1))	288	94%	5	75, 1	104.17, 0.74	41.95, 8.83

TABLE II. KERNEL FEATURES FOR S-REDUCTION.

Solution	#Reg.	Shared Memory	Block size	Theoretical Occupancy	MBPM
TB2	10	2(2B+2B/nBanks)	256	100%	-
TB4-warp	14	2(4B+32)	256	83%	-
Matrix	11	2H×(W+1)	256	83%	-
Persistent Matrix	20	2H×(W+1)	256	83%	5
DiffWarps	25	4H×(W+1)	288	38%	2
DiffWarps_{16}	21	2H×(W+1)	288	94%	5

more divergent for segmented reduction.

With regard to the other scenarios, tree-based solutions are gaining positions as the segment size gets smaller. They overtake `Diffwarps` and $\texttt{Diffwarps}_{16}$ in case (b), and exhibit the best performance in case (c). The reason we find is that the probability of getting coalesced accesses, concerning irregular accesses, increases for tree-based reductions when segments are small. To explain it let us focus on segments of size 3, i.e. case (c). We have simulated the first launch of `TB2`, `TB4-warp` and `Matrix` to analyze the ratio of read data transfers to requested accesses. Transfers have been counted according to the way Fermi serves memory in chunks of 32 adjacent floats (128 bytes). Table IV examines what happens to the first CUDA-warp of the first CUDA-block. According to the values of *D* for each solution, `TB2` runs 9 recursive levels, `TB4-warp` requires 5 levels for intra-warp reductions, and `Matrix` processes 32 columns. Since the results of columns 1, 2 and 3 get repeated, only columns from 0 to 6 are presented in the table. In addition, `TB4-warp` and `Matrix` require 5 levels more to reduce intermediate results. They have been included in the table for `TB4-warp` (levels 5-9). Notice that `TB2` exhibits the lowest ratios, especially in the first levels, which indicates that irregular accesses are quite coalesced. On the contrary, the ratio for `Matrix` is always 1, that is, each request access is served with an independent transfer, which penalizes its performance.

VI. CONCLUSION AND FUTURE WORK

Sequential approaches have a better performance than tree-based ones for unsegmented reduction. With regards to the segmented case, performance depends on the distribution of segments. According to our results for regularly-spread segments, tree-based methods exhibit a higher bandwidth for small sizes, whereas sequential ones run faster for medium and large ones.

The two optimizations we have presented result in a speed-up for the unsegmented problem. Concerning the segmented case, performance is improved by using persistent blocks over segments of large size, while it remains the same for medium/small segments. `Diffwarps` is very shared-memory demanding, which makes its occupancy decrease. Thus, it is not competitive for segmented reduction on nowadays graphics hardware, although this could change for future devices since the current tendency has been to increase shared memory size.

The optimizations only have been integrated into the sequential matrix-based algorithm, and we plan to test them onto the tree-based methods. Finally, the reduction is a part of the scan primitive and the optimizations presented in this paper could be embedded in the scan algorithms to improve their performance.

TABLE III. PROFILER'S REPORT FOR $30*2^{20}$ ELEMENTS.

Solution	#Blocks	Divergent Branches (%)	
		Unsegmented	One Segment
TB2	61440, 120, 1	0.56, 0.70, 0.56	2.74, 2.74, 2.704
TB4-warp	30720, 30, 1	0.42, 0.42, 0.42	24.77, 24.81, 24.81

TABLE IV. ANALYZING IRREGULAR ACCESSES IN SEGMENTS OF SIZE 3. ONLY THE FIRST CUDA-WARP IS CONSIDERED.

TB-level/Matrix-column		0	1	2	3	4	5	6	7	8	9
TB2	accesses	11	21	32	32	16	8	4	2	1	-
	transfers	1	2	3	6	6	6	4	2	1	-
	trans./acc.	0.09	0.09	0.09	0.18	0.37	0.75	1	1	1	-
TB4-warp	accesses	5	5	4	2	1	16	8	4	2	1
	transfers	1	1	1	1	1	11	8	4	2	1
	trans./acc.	0.2	0.2	0.25	0.5	1	1	1	1	1	1
Matrix	accesses	11	10	11	11	10	11	11	…	…	…
	transfers	11	10	11	11	10	11	11	…	…	…
	trans./acc.	1	1	1	1	1	1	1	…	…	…

ACKNOWLEDGMENT

Research supported by the Spanish Projects CCG10-UCM/TIC-5476 and GR35/10-A-921547.

REFERENCES

[1] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," Proc. High Performance Graphics (HPG 09), ACM, 2009, pp. 145–149, DOI=10.1145/1572769.1572792.

[2] M. Bauer, H. Cook, and B. Khailany, "*CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization*," Proc. Int. Conference for High Performance Computing, Networking, Storage and Analysis (SC '11), ACM, November 2011, DOI=10.1145/2063384.2063400.

[3] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," NVIDIA TR NVR-2008-004, Dec. 2008.

[4] G. E. Blelloch, Vector Models for Data-Parallel Computing, MIT Press, 1990.

[5] G. E. Blelloch, "Prefix sums and their applications," Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1993.

[6] CUDA data parallel primitives library (CUDPP). http://gpgpu.org/developer/cudpp.

[7] Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," Proc. *international conference on Supercomputing* (ICS 08), ACM, 2008, pp. 205-213, DOI=10.1145/1375527.1375559.

[8] General-Purpose Computation on Graphics Hardware http://gpgpu.org/

[9] A. Gress, M. Guthe and R. Klein, "GPU-based collision detection for deformable parameterized surfaces," Computer Graphics Forum 25, 2006, pp. 497–506, DOI: 10.1111/j.1467-8659.2006.00969.x.

[10] M. Harris, Optimizing Parallel Reduction in CUDA, (2007), http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf.

[11] D. Horn, "Stream reduction operations for GPGPU applications," GPU Gems 2, M. Pharr (ed.), Addison Wesley, 2005, pp. 573–589.

[12] P. J. Martín, R. Torres, and A. Gavilanes,. "CUDA Solutions for the SSSP Problem," Proc. International Conference on Computational

Science (ICCS 09), Springer-Verlag, 2009, pp. 904-913, DOI=10.1007/978-3-642-01970-8_91.

[13] S. Sengupta, A. E. Lefohn and J. D. Owens, "A work-efficient step-efficient prefix sum algorithm," Proc. Edge Computing Using New Commodity Architectures, 2006, pp. 26–27.

[14] S. Sengupta, M. Harris, Y. Zhang, and J. Owens, "Scan Primitives for GPU Computing," Proc. Graphics Hardware (GH 07), ACM, 2007, pp. 97-106.

[15] S. Sengupta, M. Harris, M. Garland, "Efficient Parallel Scan Algorithms for GPUs," NVIDIA TR NVR-2008-003, Dec. 2008.

[16] W. Wang, Y. Huang and S. Guo, "Design and Implementation of GPU-Based Prim´s Algorithm," Modern Education and Computer Science, 4, MECS-Press, 2011, pp 55-62.

[17] K. Zhou, Q. Hou, R. Wang and B. Guo, "Real-time kd-tree construction on graphics hardware," Proc. ACM SIGGRAPH Asia 2008, ACM, 2008, pp. 1-11, DOI=10.1145/1457515.1409079.

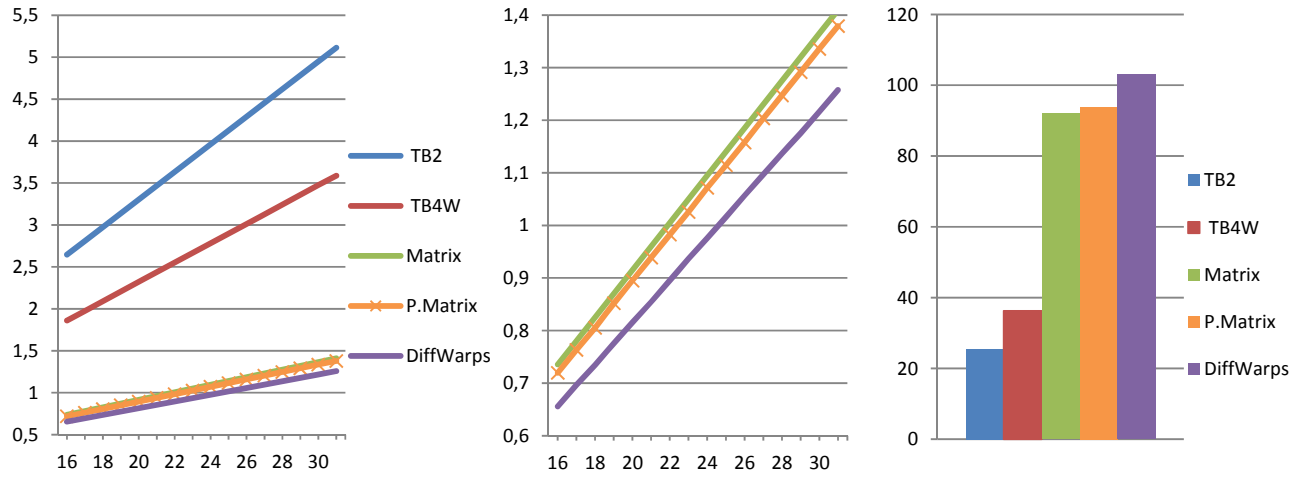
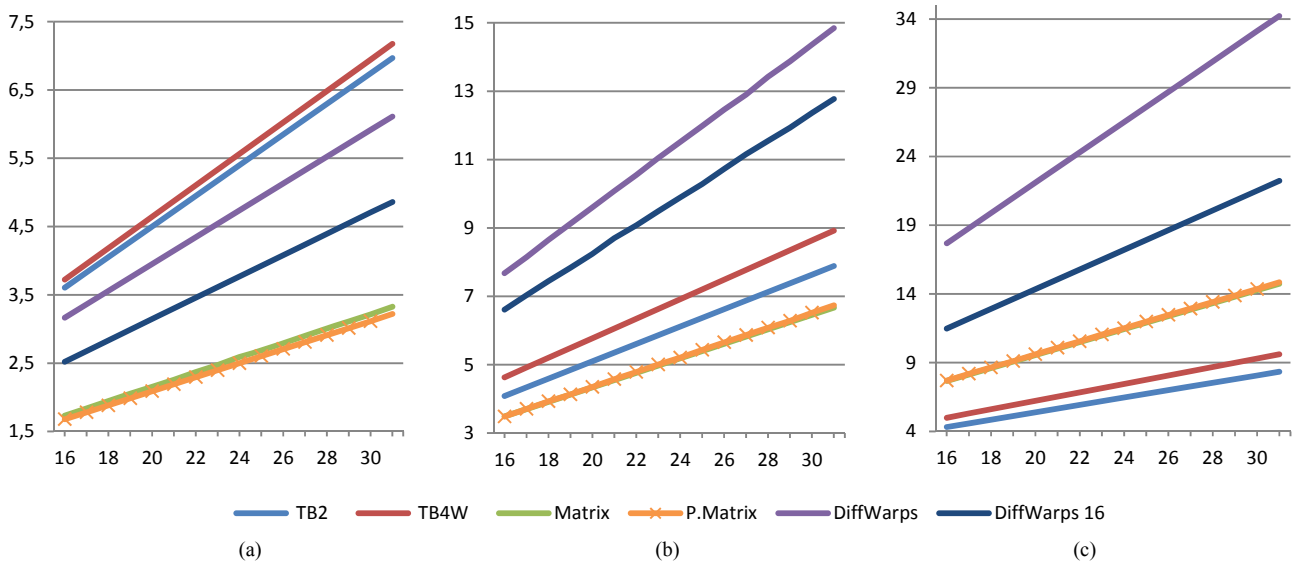


Figure 4. Results for unsegmented reduction. (Left and center) runtimes in ms (Y-axis). Input size is x*2^{20} elements (X-axis). (Right) effective bandwidths (GB/s).



(a) (b) (c)

Figure 5. Runtimes in ms (Y-axis) for s-reduction. Input size is x*2^{20} elements (X-axis). (a) Only one segment, (b) random sized segments, (c) segments of size 3.

CUDA solutions for the SSSP problem*

Pedro J. Martín, Roberto Torres, Antonio Gavilanes
Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain
{pjmartin@sip, r.torres@fdi, agav@sip}.ucm.es

Abstract. We present several algorithms that solve the single-source shortest-path problem using CUDA. We have run them on a database, composed of hundreds of large graphs represented by adjacency lists and adjacency matrices, achieving high speedups regarding a CPU implementation based on Fibonacci heaps. Concerning correctness, we outline why our solutions work, and show that a previous approach [10] is incorrect.

Keywords: Shortest path algorithms, GPU, CUDA.

1 Introduction

Computing shortest paths in a graph is one of the most fundamental problems in computer science and network optimization. In particular, the *Single-Source Shortest-Paths* (in the sequel, SSSP) problem, which computes the weight of the shortest path from a specific vertex (source) to all other vertices, in a weighted directed graph, is a heavily studied problem in graph theory.

Probably, the most well-known algorithm solving this problem for the case of graphs with nonnegative edges was given by Dijkstra in 1959 [1], and nearly all the subsequent proposals are based on it. In spite of its early formulation, this classic solution is still presented in almost every textbook on algorithms [2]. After the simplest Dijkstra´s implementation, which uses arrays to represent min-priority queues and runs in $O(n^2)$ time, where n is the number of vertices, many authors have designed different data structures to implement these queues and achieve better and better asymptotic running times. In particular, Fibonacci heaps [3] can be used to get $O(m + n \log n)$, where m is the number of edges.

As [4] points out, Dijkstra's algorithm is inherently sequential since its efficiency depends on a fixed ordering of the vertices. The Bellman-Ford algorithm allows all vertices to be considered in parallel but at the cost of being not efficient. Different formulations of parallel algorithms for the SSSP problem are reviewed in detail in [5]. In particular, a specific proposal for incorporating parallelism into Dijkstra's algorithm has been the introduction of parallel priority queues [6]. However, the literature contains few experimental studies on parallel algorithms of the nonnegative SSSP problem. Some of the more recent works study the use of supercomputers for solving large graphs. [7] reports performance results on the multithread parallel computer Cray MTA-2, using the Δ-stepping parallel algorithm of [5]. [7] exhibits remarkable parallel speedup when compared to competitive sequential algorithms, for low-diameter sparse graphs of 100 million vertices and 1 billion edges. On the other hand, [8] contains an experimental evaluation of [6] on the APEmille supercomputer, but restricted to graphs with no more than thousands of vertices.

* Research supported by the Project CCG08-UCM/TIC-4252.

However, some modern applications, such as data mining, network organization, etc. require large graphs with millions of vertices, and some of the previous algorithms become impractical, when we do not have a very expensive hardware at our disposal. Fortunately, Graphics Processing Units (GPUs) supply a high parallel computation power at a low price. Moreover, they have become very popular since the languages involved in their programming have evolved from graphics APIs to general purpose languages. One of the best examples is the CUDA API [9] of NVIDIA. As a consequence of this evolution, the so called General Purpose Computing on GPU (GPGPU) [11] has consolidated as a very active research area, where many problems that are not directly related to computer graphics are solved using GPUs. The aim of all these GPU-based implementations is to achieve better running times than their CPU-based counterparts.

With this new technology available, the natural challenge is: "how can GPUs be used to solve the SSSP problem?". Unfortunately programming with CUDA must be carefully taken, basically because CUDA programming model is very restricted concerning synchronization, and the unique proposal we are aware of ([10]) is not correct. Apart from giving a counterexample, in this paper we present different correct solutions, based on Dijkstra´s algorithm, that are experimentally compared using a database of hundreds of randomly generated large graphs.

2 Dijkstra´s algorithm overview

Dijkstra´s algorithm solves the SSSP problem for directed graphs $G = (V, E)$ in which every edge $(v, v') \in E$ has a positive weight $\omega(v, v') > 0$. Let n and m be the number or vertices and edges respectively. We assume that vertices are numbered from 0 to $n - 1$, and that 0 is the source vertex. The algorithm splits the set of vertices in two parts: the set R of *resolved vertices* (R-vertices) and the set U of *unresolved vertices* (U-vertices), and it keeps a shortest-path estimate $c[i]$ for each vertex i, which actually coincides with the shortest path weight for R-vertices. For U-vertices, $c[i]$ holds the weight of the *shortest special path* (SSP) to i w.r.t. R, that is, the shortest path among the paths to i that exclusively traverses R-vertices before reaching i.

The algorithm implements a loop. Each iteration is composed of three steps: (1) the estimates for U-vertices are relaxed using the last vertex added to R, which we will call the *frontier vertex*, (2) the minimum estimate for U-vertices is computed, and (3) a U-vertex with the minimum estimate is promoted to R, and becomes the new frontier. Figure 1 presents a typical Dijkstra's algorithm implementation that includes the variable `f` to hold the current frontier vertex. Regardless of the graph representation we chose, it runs in $O(n^2)$.

The soundness of the algorithm is based on two fundamental properties that can be proved. First, anytime a new frontier arises in the third step, its estimate actually coincides to the weight of its shortest path, thus it can be safely promoted to R. Second, in order to relax the estimates of a U-vertex j using the current frontier vertex f, the SSP to j w.r.t. R cannot traverse more R-vertices after visiting f, hence we only consider the previous estimate and $c[f] + \omega(f, j)$ when updating $c[j]$.

```
void Dijkstra (c) {
  forall vertex i { c[i]=INFINITY; u[i]=true;}
  c[0]=0; u[0]=false;
  f=0; mssp=0;
  while (mssp!=INFINITY) {
    forall unresolved vertex j {
         c[j]= min(c[j], c[f]+w[f,j]);}
    mssp= INFINITY;
    forall unresolved vertex j
        if(c[j]<mssp){ mssp=c[j]; f=j;}
    u[f]=false;
  }//while
}
```

Fig. 1. Dijkstra's algorithm implementation.

3 Parallelizing Dijkstra's algorithm

Dijkstra´s algorithm handles a unique frontier vertex even when the estimates of several U-vertices coincide with the minimum computed in the second step. In these cases, the algorithm simply chooses one of them to compose the new frontier. In consequence, it requires a different iteration to promote each of them to R. Fortunately, this set of U-vertices, which we will call F, can be processed at once because the previous two properties remain:

1. Their estimates actually coincide with the weight of their shortest paths.
2. In order to relax the estimate of a remaining U-vertex j, the SSP to j w.r.t. R cannot traverse more R-vertices after visiting one F-vertex, hence only the previous estimate and $\min_{f \in F}\{c[f] + \omega(f,j)\}$ must be considered when updating $c[j]$. In particular, note that only one F-vertex can belong to the SSP to j w.r.t. R.

Therefore the notion of *compound frontier* can be used to design the *Dijkstra's algorithm Adapted to Compound Frontiers* (DA2CF) presented in Fig. 2. Although the algorithm is composed of the same three basic operations, their implementations must suitably handle compound frontiers:

1. `relax(c, f, u)` must relax the shortest path estimate for every U-vertex using F-

```
void DA2CF(c) {
  initialize(c, f, u);
  mssp = 0;
  while (mssp != INFINITY) {
    relax(c, f, u);
    mssp = minimum(c, u);
    update(c, f, u, mssp);
  }//while
}
```

```
void initialize(c, f, u) {
  forall vertex i {
    c[i] = INFINITY;
    f[i] = false;
    u[i] = true;
  }//for
  c[0] = 0;
  f[0] = true;  u[0] = false;
}
```

Fig. 2. Dijkstra's algorithm adapted to compound frontiers.

vertices. Hence it must compute $c[j] = min\{c[j], c[f] + \omega(f,j)\}$ for every pair of vertices $j \in U$ and $f \in F$.

2. `minimum(c, u)` must find the minimum estimate of the U-vertices, called `mssp`.
3. `update(c, f, u, mssp)` must update the set of U-vertices by removing those vertices whose estimate is equal to `mssp`, which will compose the new set of F-vertices.

There are many ways to implement these operations. Although sequential solutions could be easily written by means of the obvious single loop (two nested loops for the `relax` procedure), the operations can be performed in parallel, by launching a thread for each iteration of the loop (the main loop for `relax`).

Figure 3 shows two versions of the `relax` procedure. On the left, `relax_F` processes F-vertices: "for each F-vertex we visit all of its successors, relaxing c for those vertices that are still unresolved". Observe that the sentence `c[j]=min(c[j], c[i]+w(i,j))` could produce concurrency inconsistencies if two F-vertices `i` and `i'` accessed the same U-vertex `j` and the worst value `c[i]+w(i,j)` were finally left. In order to prevent such inconsistencies, we use the atomic instruction `atomicMin(x,y)` that allows only one thread to store the minimum of `x` and `y` in the variable `x`.

CUDA devices of compute capability 1.0 do not support atomic functions, thus we propose another approach that does not use them. Figure 3 on the right presents the `relax_U` procedure which focuses on U-vertices instead of F-vertices: "for each U-vertex, we visit all of its predecessors, relaxing its c-value when a F-vertex is found". Notice that this approach requires predecessors instead of successors.

A parallel version of the `minimum` function is a more difficult task, because of its sequential nature. Fortunately, different reduction procedures have been already adapted to the stream model [12, 13, 14]. In this paper we have adapted the `reduce3` method included in the CUDA SDK 1.1 [15] to obtain the `minimum1` procedure of Fig. 4 on the right.

Finally, we parallelize the `update` procedure as Fig. 4 on the left shows. In the sequel, DA2CF_F and DA2CF_U will denote the algorithms that use `relax_F` and `relax_U`, respectively.

Regarding asymptotic complexity, let us compare the Dijkstra´s algorithm of Fig. 1, that runs in $O(n^2)$, to the sequential versions of the DA2CF algorithm that result when the "in parallel" qualifier is erased. Firstly, notice that the number of iterations required for the main DA2CF-loop depends more heavily on the given graph; since

```
void relax_F(c, f, u) {
  forall i in parallel do {
    if (f[i]) {
      forall j successor of i do {
        if (u[j])
          atomicMin(c[j],c[i]+w[i,j]);
      }//for
    }//if
  }//for
}
```

```
void relax_U(c, f, u) {
  forall i in parallel do {
    if (u[i]) {
      forall j predecessor of i do {
        if (f[j])
          c[i]= min(c[i],c[j]+w[j,i]);
      }//for
    }//if
  }//for
}
```

Fig. 3. Processing frontier (left) or unresolved (right) vertices within the `relax` operation.

```
void update(c, f, u, mssp) {
  forall i in parallel do {
    f[i] = false;
    if (c[i] == mssp) {
      u[i] = false;
      f[i] = true;
    }//if
  }//for
}
```

```
void minimum1(u, c, minimums) {
  forall i in parallel do {
    thid = threadIdx.x;
    i = blockIdx.x*(2*blockDim.x)+threadIdx.x;
    j = i + blockDim.x;
    data1 = u[i] ? c[i] : INFINITY;
    data2 = u[j] ? c[j] : INFINITY;
    sdata[thid] = min(data1, data2);
    __syncthreads();
    for (s = blockDim.x/2; s>0; s>>=1) {
      if (thid<s) {
        sdata[thid]=min(sdata[thid],sdata[thid+s]);
      }// if
      __syncthreads();
    }// for
    if (thid==0) minimums[blockIdx.x]= sdata[0];
  }// forall
}
```

Fig. 4. Updating the frontier (left), and computing the minimum sssp with CUDA (right).

the size of the arising compound frontiers influences its termination. Hence, we analyze its worst case. We focus on adjacency lists since they fit better to large graphs and they provide the algorithm of Fig. 1 with smaller running times. The worst case corresponds to a complete graph requiring n iterations (the frontier size is always 1), DA2CF_F also takes a time in $O(n^2)$, but with a greater constant due to the management of the `f` array. However, DA2CF_U takes a time in $O(n^3)$, since the edges arriving at an unresolved vertex are repeatedly processed while it remains unresolved. In order to evaluate the general case, we experimentally run CUDA implementations on randomly generated graphs.

4 CUDA implementations

The adjacency list representation of a graph is made up of three arrays: v for vertices, e for edges and ω for weights. Array v is used to access the adjacency list of each vertex. Specifically, the adjacency list of the vertex i appears in e and ω from index $v[i]$ to index $v[i+1]-1$ (Fig. 5 on the left). In order to deal with the last vertex in the same way, an extra component is added at the end of v such that $v[n]=m$. In consequence, array v is of size $n+1$ and both e and ω are of size m.

There are two possible interpretations for the data occurring in e. Vertices belonging to the adjacency list of vertex i can be understood as successors or predecessors. Formally, in the predecessor interpretation, there is an edge to i from each adjacent vertex, whereas in the successor interpretation the edge goes from i to each adjacent vertex. Graphs must be represented in the proper interpretation before execution, since the `relax` procedure requires either successors (`relax_F`) or predecessors (`relax_U`), but not both.

4.1 Implementations

We have sequential C implementations corresponding to the sequential versions of DA2CF_F and DA2CF_U, that we respectively call F^{CPU} and U^{CPU}. Before presenting the pure CUDA implementations, we have tried some hybrid systems running on both, CPU and GPU. Since the `minimum` function is inherently sequential, we have restricted this function to run on CPU. Moreover, in order to fit the requirements of any CUDA device, we have focused on the `relax_U` procedure. Hence, we have designed three hybrid implementations based on the DA2CF_U algorithm: U^{H1}, U^{H2} and U^{H3} which respectively run the `update` procedure, the `relax_U` procedure, and both `update` and `relax_U` on the GPU.

In order to run the complete algorithm on GPU, we must run additional passes of the `minimum` function, since the `minimum1` kernel of Fig. 4 only reduces each block to a single value. Thus, we have implemented another kernel, called `minimum2`, to execute a second pass on GPU. The obtained values are finally minimized on CPU in a sequential manner, because the number of these values is too small. Hence, we have two fully GPU-implemented solutions based on the DA2CF_U algorithm, U^{GPU} and $U^{GPU+2min}$ that apply one and two minimization passes, respectively. Based on the DA2CF_F algorithm, we also have two fully-GPU solutions, but this time they have been designed to analyze the cost due to simultaneous accesses to the `c` array. Thus, apart from the F^{GPU} solution, we have another one, called $F^{GPU\_no\_Atomic}$, that does not use the atomic function `atomicMin` but a non-atomic function `min`. We introduce the latter solution only for measuring purposes, since it is not correct in a parallel environment. Anyway, both solutions apply a single `minimum` pass.

4.2 Exploiting CUDA resources

It is possible to accelerate the U^{GPU} solution by using some CUDA features. Concretely, in this subsection we exploit texture cache and shared memory to improve the implementation of the `relax_U` kernel. Let us call the corresponding solution U^{GPU_PLUS}.

In order to retrieve the boundaries of the adjacency list, the *i*–th thread must access $v[i]$ and $v[i+1]$, whereas the $(i+1)$–th thread must access $v[i+1]$ and $v[i+2]$. Thus, the value $v[i+1]$ is shared by the two threads, and can be brought

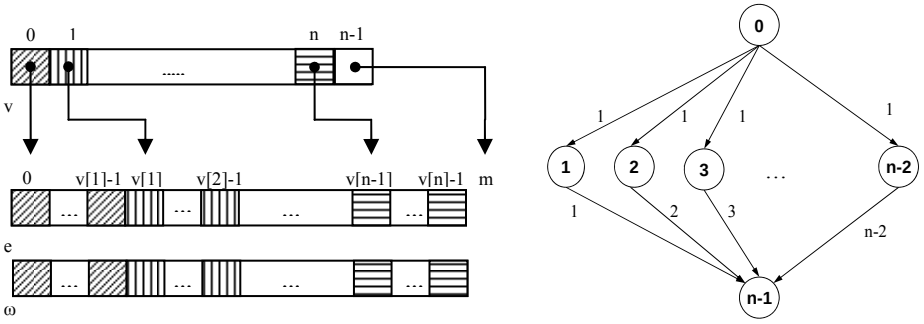


Fig. 5. Left: The adjacency list representation. Right: Counterexample to [10].

only once if shared memory is used. Then, each thread i reads $v[i]$ from global memory, writes it to shared memory, and after that, it reads $v[i+1]$ from shared memory directly. A special case is the last thread of a block, since it will bring both $v[i]$ and $v[i+1]$.

Notice that two threads can access the array `f` for the same vertex `j`. To accelerate the corresponding readings, the array can be accessed through a texture, taking advantage of the texture cache. Thus, it is possible for threads of the same block to read `f[j]` from the cache and not from global memory.

4.3 A bugged implementation

Let us explain the problem we have found in the solution presented in [10]. The authors propose an implementation of Dijkstra´s algorithm which relaxes using the frontier, in a similar way to our `relax_F` procedure. However, instead of the atomic function `atomicMin`, they use the following code to relax a vertex `j` which is successor of a frontier vertex `i`:

```
if (uc[j] > c[i]+w[i,j])  uc[j] = c[i]+w[i,j];
```

where the array `uc`, called the *updating cost array*, holds a copy of the array `c` before relaxing. Indeed, as the authors explain, the new cost is not reflected in `c`, but is updated in `uc`, in order to avoid read-after-write inconsistencies. Later, they dump `uc` onto `c` in the `update` kernel.

Unfortunately, this technique is not enough to avoid write-after-write inconsistencies. Concretely, if two frontier vertices `i` and `i'`, whose related threads are simultaneously running, satisfy `uc[j]>c[i]+w[i,j]` and `uc[j]>c[i']+w[i',j]` at the same time for the same U-vertex `j`, then both threads will make the above *if*-condition true. Thus, there will be no control on the final value assigned to `uc[j]`. Since debugging concurrent programs is highly difficult, we have defined the graph of Fig. 5 on the right in order to increase the number of these critical situations.

We have run their implementation on a GeForce 8800 GTS, similar to the GeForce 8800 GTX used in [10], with n=1024 and 32 threads per block. Furthermore, we have also tested the undirected version of the graph, since the authors do not specify the kind of graph they manage. In any case, observe that vertices ranging from 1 to $n-2$ will compose the frontier after the first iteration. Actually, $c[i] = uc[i] = 1$, for $1 \leq i \leq n-2$, and $c[n-1] = uc[n-1] =$ INFINITY, after the first iteration. Hence, every vertex i $(1 \leq i \leq n-2)$, tries to relax $uc[n-1]$ to a different value during the second iteration. In consequence $c[n-1]$ ends with a value that randomly changes from execution to execution, instead of computing the right solution $c[n-1] = 2$.

Since threads of different blocks cannot be synchronized in CUDA, solving this bug requires the use of atomic functions in the `relax_F` implementation. Unfortunately such functions are only available from compute capability 1.1, so solving this bug for the cards GeForce 8800 GTS and GTX demands a deeper modification of the algorithm. This is actually the aim of our `relax_U` kernel.

5 Adjacency matrices

In the case of adjacency lists, it is difficult to conceive a method to allow threads to collaborate when reading from global memory. On the opposite, when adjacency

matrices are used, threads must visit every element of each column or row, and so, threads can cooperate to bring elements of arrays `f`, `c` or `u` to shared memory.

As we did for the adjacency list representation, we can consider two kind of implementations: one that looks for predecessors (U^{CPU} and U^{GPU}), and another one that looks for successors (F^{CPU} and F^{GPU}). In U^{GPU}, each thread t must look for its predecessors by visiting the t–th column. In order to make threads collaborate, the exploration is divided in chunks of b elements, where b is the number of threads in a block. The arrays `f` and `c` are also divided in chunks of b elements. Before visiting the chunk of the column, each thread brings a component of the chunk of `f` and `c` into shared memory. That way, the information of the arrays `f` and `c` is already available when each predecessor within the chunk is processed. Once a chunk is dispatched, the next one is processed identically.

On the other hand, F^{GPU} processes frontier vertices, so each thread explores a row. Threads can also collaborate similarly, but this time they bring elements of `u`.

6 Results and discussion

We have tested all the implementations using an Intel CORE 2 QUAD Q6600 2.40 GHz 2GB DDR memory, and a NVIDIA GEFORCE GTX 280, which has 30 multiprocessors and 1 GB of GDDR3 memory, using 256 threads per block. The database is composed of randomly generated graphs with a number of vertices that ranges from 1 to 11 M for adjacency lists, and from 1 to 15 K for adjacency matrices. The database includes 25 graphs for each of these sizes. The degree of each graph is fixed, so every vertex has the same number of adjacent vertices. The chosen degree is 7 for adjacency lists, while $n/5$ for matrices. Concerning lists, graphs have been generated using the predecessor interpretation, so we have also turned each graph into its successor interpretation. Notice that the degree of the graph can not to be kept after this operation. Edge weights are integers that randomly range from 1 to 10.

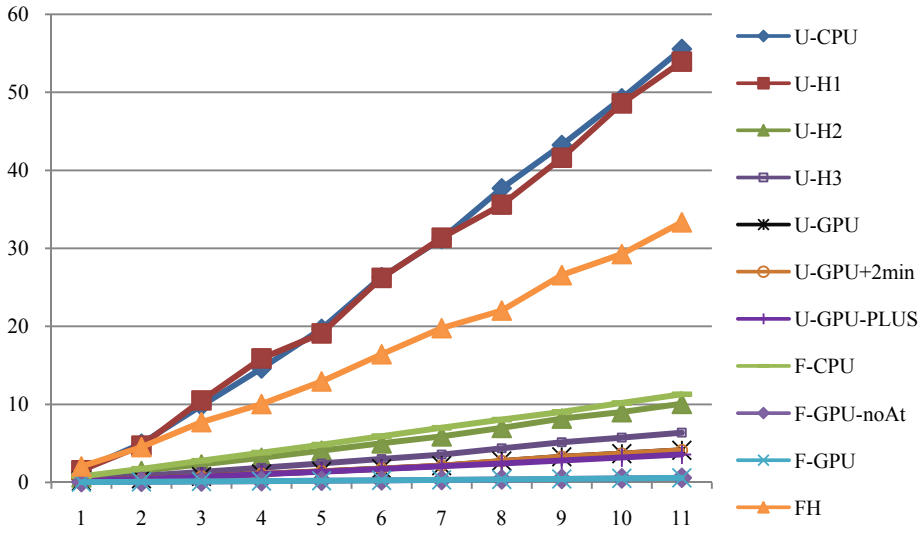


Fig. 6. Results for adjacency lists. Units: seconds for the y-axis and 2^{20} vertices for the x-axis.

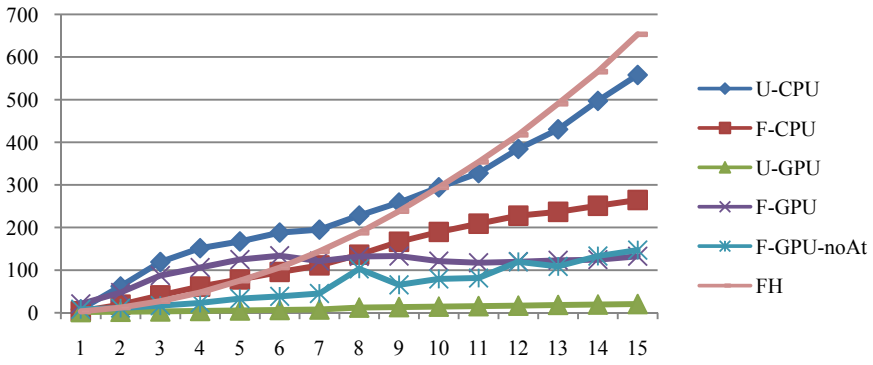


Fig. 7. Results for adjacency matrices. Units: ms. for the y-axis and 2^{10} vertices for the x-axis.

Figures 6 and 7 show the results we have obtained comparing the average times for each solution to a CPU-solution, called FH, implemented using Fibonacci Heaps and based on the SPLIB library [16]. Most of our solutions, including some CPU ones, run faster on these graphs since the arising frontiers are large. Thus, our solutions only require a few iterations to solve the problem. Concretely, around 45 are enough to solve the largest graphs represented with adjacency lists.

Let us analyze the results for the adjacency list representation presented in Fig. 6. The figure shows that fully CUDA-implemented solutions (U^{GPU}, $U^{GPU+2min}$ and F^{GPU}) are more efficient than partially CUDA-implemented ones (U^{H1}, U^{H2} and U^{H3}), which is due to the overhead connected to the data movement between CPU and GPU. The figure also shows that a two-pass minimization behaves as a single one, since U^{GPU} and $U^{GPU+2min}$ overlap. This can be explained comparing the number of values provided by `minimum1` to those provided by `minimum2`. Notice that these numbers are $n/(2b)$ and $n/(2b)^2$, respectively, where b is the number of threads per block. Since n ranges from $1 * 2^{20}$ to $11 * 2^{20}$ and we have chosen $b = 256 = 2^8$, these numbers finally range from 2^{11} to $11 * 2^{11}$ for `minimum1` and from 2^2 to $11 * 2^2$ for `minimum2`. Therefore, the number of values that must be copied from GPU to CPU, in order to be minimized on CPU, is similar for U^{GPU} and $U^{GPU+2min}$; so there is no difference in time consumption. The figure also shows that exploiting CUDA resources leads to better results, since U^{GPU_PLUS} is slightly faster, obtaining a factor near 10X w.r.t. FH.

Concerning solutions based on the `relax_F` procedure, Fig. 6 shows that processing unresolved vertices is slower than processing the frontier, even for parallel implementations, since F^{GPU} is quite faster than U^{GPU}. Also notice that $F^{GPU\_no\_Atomic}$ behaves as F^{GPU} because the simultaneous accesses to the same c-component are rare when the degree is small. These solutions reach a factor around 60X w.r.t. FH.

Regarding adjacency matrices (Fig. 7), the more vertices the graph has, the higher is the degree. Thus, the frontier sets are huge, and `relax_F` based solutions are slower than `relax_U` based ones. To summarize, U^{GPU} is the fastest, achieving a factor of 32X w.r.t. FH. Finally, the figure gives more insight about how atomic operations affect the overall performance, since $F^{GPU\_no\_Atomic}$ is usually faster than F^{GPU}.

7 Conclusions

GPUs can be used to speed up solutions to many problems, including classic problems. Nevertheless, the CUDA programming model is very restricted concerning synchronization, so implementations must be carefully designed, and intuitions about their correctness should be given at least.

In the paper we have shown different CUDA solutions for the SSSP problem, considering adjacency lists and matrices. We have also explained the bug we found in [10], which is basically due to write-after-write inconsistencies. In order to solve this bug, two approaches have been shown. On the one hand, atomic functions can be used for devices of compute capability 1.1 and higher. On the other one, the usual `relax` procedure can be reversed in order to process unresolved vertices instead of frontier vertices. Although processing unresolved vertices is theoretically less efficient, the latter approach is the only applicable solution to any CUDA device.

References

1. Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. Num. Math. 1 (1959), 269-271.
2. Cormen, T., Leiserson, C., Rivest, R. and Stein, C. 2001. Introduction to algorithms (second edition). The MIT Press.
3. Fredman, M. L. and Tarjan, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM 34 (1987), 596-615.
4. Meyer, U. and Sanders, P. 1998. Δ-stepping: a parallel single source shortest path algorithm. In Proc. ESA'98. LNCS 1461, 393-404.
5. Meyer, U. and Sanders, P. 2003. Δ-stepping: a parallelizable shortest path algorithm. J. of Algorithms 49 (2003), 114-152.
6. Brodal, G., Träff, J. and Zaroliagis, C. 1998. A parallel priority queue with constant time operations. J. Parallel and Distributed Computing 49 (1998), 4-21.
7. Madduri, K., Bader, D., Berry, J. and Crobak, J. 2006. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In Proc. Workshop on Algorithm Engineering and Experiments (ALENEX'07).
8. Di Stefano, G., Petricola, A. and Zaroliagis, C. 2006. On the implementation of parallel shortest path algorithms on a supercomputer. In Proc. ISPA 2006. LNCS 4330, 406-417.
9. http://www.nvidia.com/object/cuda_home.html#
10. Harish, P. and Narayanan, P. J. 2007. Accelerating large graph algorithms on the GPU using CUDA. In Proc. HiPC 2007. LNCS 4873, 197-208.
11. http://www.gpgpu.org/
12. Buck, I. and Purcell, T. 2004. A toolkit for computation on GPUs. In GPU Gems, Chapter 37. Addison-Wesley.
13. Harris, M., Sengupta, S. and Owens, J. 2008. Parallel prefix sum (Scan) with CUDA. In GPU Gems 3, Chapter 39. Addison-Wesley.
14. Harris, M. 2007. Parallel prefix sum (Scan) with CUDA. http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/scan/doc/scan.pdf
15. http://www.nvidia.com/object/cuda_get.html
16. http://avglab.com/andrew/soft.html

Soluciones CUDA al problema del camino más corto desde un solo origen

P. J. Martín de la Calle, R. Torres de Alba, A. Gavilanes Franco

Departamento de Sistemas Informáticos y Computación, Universidad Complutense, Madrid, España
{pjmartin@sip.,r.torres@fdi.,agav@sip.}ucm.es

Resumen

Las GPUs han evolucionado hasta convertirse en máquinas altamente paralelizables con una capacidad de cómputo enorme. Además, actualmente pueden conseguirse a bajo precio, por lo que constituyen un hardware del que disponen casi todos los ordenadores corrientes. Aunque los algoritmos de búsqueda sobre grafos pueden paralelizarse, se han diseñado pocos algoritmos paralelos para esta tarea, ya que no existía a mano hardware real de este tipo. En este trabajo presentamos una serie de algoritmos que resuelven en GPU el problema del camino más corto desde un solo origen para grafos de gran tamaño, usando CUDA en su implementación. Probamos la corrección de todos ellos, mostrando además que la propuesta de [10] es incorrecta. En cuanto a ejecución, hemos obtenido tiempos que son hasta 15 veces mejores que sus correspondientes en CPU para grafos dispersos representados mediante listas de adyacencia, con 8 millones de vértices y 56 millones de arcos; y hasta 35 veces mejores para grafos densos representados por matrices de adyacencia, con 3 mil vértices y 1 millón y medio de arcos.

1. Introducción

Calcular caminos mínimos en un grafo es uno de los problemas fundamentales en áreas de la informática como la optimización de redes. En particular, el problema del camino más corto desde un solo origen (lo que abreviaremos por sus siglas en inglés *S(ingle)-S(ource) S(hortest)-P(ath) problem*), que calcula el coste del camino más corto desde un vértice específico (origen) a todos los vértices de un grafo dirigido y valorado, ha sido uno de los problemas más ampliamente estudiado en teoría de grafos.

Probablemente, el algoritmo más conocido, que resuelve este problema para el caso de grafos dirigidos con valores no negativos en sus arcos, sea el que dio Dijkstra en 1959 [1]. De hecho, en él se basan casi todas las propuestas posteriores y, a pesar de haber sido formulado hace tanto tiempo, continúa presente en la mayoría de los libros sobre algoritmos [2]. La implementación original de Dijkstra estaba basada en matrices de adyacencia y su tiempo de ejecución era $O(n^2)$, donde n es el número de vértices. Posteriormente se diseñaron diferentes estructuras de datos con las que era posible alcanzar mejores tiempos. En particular, los montículos de Fibonacci [3] permiten rebajar la complejidad a $O(m + n \log n)$, donde m es el número de arcos.

Como señala [4], el algoritmo de Dijkstra es inherentemente secuencial puesto que trabaja sobre un orden fijo de los vértices. Por su parte, el algoritmo de Bellman-Ford, que también resuelve este problema, permite tratar todos los vértices de forma paralela, pero a costa de ser más ineficiente que el primero. Se han dado formulaciones paralelas para el problema SSSP (cfr. [5] para más detalles). Por ejemplo, introduciendo colas con prioridad paralelas [6]. Sin embargo, la literatura apenas tiene estudios experimentales de algoritmos paralelos para el problema SSSP no negativo. Uno de los más recientes es [7], donde se utiliza un computador paralelo multihilo (Cray MTA-2) para resolver el problema sobre grafos enormes, mediante el algoritmo paralelo "Δ-stepping" [5]. Este algoritmo muestra una rapidez notable cuando se compara con algoritmos secuenciales competitivos, pero necesita operar sobre grafos dispersos de diámetro bajo, con 100 millones de vértices y 1000 millones de arcos. Por otra parte, [8] contiene una evaluación experimental de [6] sobre un supercomputador APEmille, pero para grafos no tan grandes, con tan solo unos miles de vértices.

Sin embargo, algunos de los algoritmos anteriores se vuelven poco prácticos, para algunas aplicaciones modernas como la minería de datos y la organización de redes, si no se dispone de un hardware excesivamente caro. En este contexto, las GPUs se han hecho muy populares entre las comunidades de programación paralela debido sobre todo a dos razones. Primero, a que proporcionan núcleos múltiples con un alto poder de cómputo y un ancho de banda elevado, unido a un coste bajo. Piénsese que una tarjeta de vídeo de última generación puede costar tan solo alrededor de 500$. Segundo, a que los lenguajes que se emplean en su manejo han evolucionado mucho, desde simples APIs gráficas hasta lenguajes de propósito general. Uno de los mejores ejemplos es el API de CUDA [9], desarrollado por NVIDIA como una extensión de C, que puede aprenderse con rapidez. Como consecuencia de esta evolución, la computación de propósito general sobre GPU (GPGPU) [11] se ha consolidado como un área activa de investigación, en la que muchos problemas, no relacionados directamente con la informática gráfica, se resuelven usando GPUs. En todos los casos, el objetivo de las implementaciones es común: alcanzar mejores tiempos de ejecución que las contrapartidas sobre CPU.

Con esta nueva tecnología a mano, el reto natural que nos hemos planteado ha sido: "¿cómo se pueden usar las

Artículo publicado en Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones (ANACAP) 2008.
ISBN 978-84-691-7426-5.

GPUs para resolver el problema SSSP?". Debido a que GPGPU es un área aún joven, hay muy pocas propuestas al respecto. Solo conocemos [10], donde se aplica CUDA para resolver problemas de búsqueda sobre grafos y, en particular, donde se dan versiones paralelas del algoritmo de Dijkstra. Desgraciadamente, CUDA se debe manejar con cuidado porque su modelo de programación es muy restrictivo en lo que se refiere a la sincronización, razón por la que la solución presentada en [10] no es correcta, como mostraremos con un contraejemplo. Por el contrario, este trabajo presentará diferentes soluciones correctas, junto con una comparación de todas ellas realizada sobre una base de datos compuesta por cientos de grafos generados de forma aleatoria.

El trabajo está organizado como sigue. En la sección siguiente explicamos las características especiales de la programación en CUDA, como modelo de procesamiento de *streams*. La Sección 3 presenta nociones preliminares y alguna notación. La Sección 4 expone –sobre una formulación específica del algoritmo de Dijkstra– distintas versiones secuenciales de este algoritmo, así como sus correspondientes versiones paralelas, y prueba la corrección de todas ellas. Después pasamos a analizar experimentalmente estos algoritmos. La Sección 5 se ocupa de la implementación de algoritmos usando grafos representados mediante listas de adyacencia. Aquí vemos que el intento presentado en [10] tiene un error. Por último, la Sección 6 hace el estudio usando grafos representados mediante matrices de adyacencia. El trabajo termina con una sección de conclusiones donde se comparan y analizan los resultados obtenidos.

2. El modelo de programación de CUDA

Las GPUs implementan la tubería gráfica. Debido a que se trata de un algoritmo inherentemente paralelo en el que se llevan a cabo cómputos de forma intensiva, la evolución de las GPUs ha seguido esta línea. Además, su bajo coste permite encontrarlas, hoy día, en casi todos los ordenadores personales. Para proporcionar mayor flexibilidad, algunas partes de las GPUs pueden programarse. Sin embargo, la forma en que se programaban era muy técnica, lo que obligaba a sus potenciales programadores a conocer detalles de la tubería gráfica, así como APIs del estilo de OpenGL o DirectX.

Por este motivo han aparecido tecnologías como CUDA. CUDA es un lenguaje muy similar a C que permite que los programadores usen gran parte de las GPUs sin necesidad de enfrentarse al incómodo modelo de programación de basados en el procesamiento de texturas. En CUDA se supone que existe un chip compuesto de varios multiprocesadores que trabajan en modo SIMD, de tal forma que puede haber cientos de hilos ejecutándose al mismo tiempo. Por ello CUDA es importante en aplicaciones científicas en las que muchos datos, independientes entre sí, pueden procesarse autónomamente, lanzando un hilo por cada dato. Cada hilo recibe una identificación numérica (`threadId`) que le permite saber qué debe hacer.

Sin embargo, no todo son ventajas. En primer lugar, todos los hilos ejecutan el mismo programa (*kernel*). En segundo lugar, el número de hilos debe ser conocido de antemano de manera que puedan ser tratados por lotes, agrupándolos en bloques del mismo tamaño. Cada bloque se ejecuta en un único multiprocesador. El número de bloques que caben en un multiprocesador depende de los recursos que los hilos consuman. La proporción entre el número de procesadores que están ocupados y el máximo disponible es la ocupación (*occupancy*) de cada multiprocesador. El conjunto de bloques se llama *grid*.

Además, cuando se programa en CUDA hay que prestar especial atención a las divergencias que se producen en el código fuente ya que los hilos se ejecutan en modo SIMD de 32 vías (a los paquetes de 32 hilos se les conoce como *warps*). Un uso elevado de instrucciones de ramificación (sentencias if-else) puede dar lugar a un rendimiento bajo cuando los hilos del mismo warp toman caminos diferentes, pues cada camino se ejecuta de forma serializada.

También es necesario fijarse en el tratamiento de la memoria. Hay tres tipos de memoria diferente en CUDA: global, compartida y registros. La global es random-access, off-chip y no cacheada (salvo si se accede mediante texturas); por lo que el intercambio de información con ella es tan costoso que debería restringirse al máximo, y en caso de necesidad, usar patrones apropiados para hacerlo. La memoria compartida es on-chip y la comparten todos los hilos de un bloque. Se divide en 16 bancos de 32 bits. Las lecturas y escrituras en memoria compartida son rápidas si no hay conflictos de bancos; estos se producen cuando varios hilos tratan de leer o escribir en el mismo banco.

El paso de información entre hilos está limitado a lecturas y escrituras en memoria compartida o global. Para evitar que los hilos compitan en rapidez, estos se deben sincronizar. En CUDA hay dos formas de hacerlo: con operaciones atómicas y con barreras. Las barreras se implementan con la instrucción nativa `__syncthreads()` que solo sincroniza hilos de un bloque. Por su parte, las operaciones atómicas, que pueden usarse en aquellas tarjetas que admiten capacidad de cómputo (*compute capability*) 1.1 o superior, suponen una disminución del rendimiento de la tarjeta.

3. Preliminares y notación

A lo largo de este trabajo usamos grafos dirigidos $G = (V, E)$ con coste positivo $\omega(v, v') > 0$ en todos sus arcos $(v, v') \in E$. El problema SSSP se formula como la búsqueda del camino más corto desde un vértice origen $s \in V$ a cualquier vértice $v \in V$. Para referirnos a los vértices, los numeramos de 0 a $|V| - 1$, de forma que 0 sea el vértice origen. Formalmente, el problema SSSP consiste en el cálculo de:

$$\delta(i) = \begin{cases} min\,\{\omega(p) : 0 \rightsquigarrow^p i\} & si\ hay\ camino\ de\ 0\ a\ i \\ \infty & en\ caso\ contrario \end{cases}$$

para todo vértice i, donde $0 \rightsquigarrow^p i$ indica que p es un camino de 0 a i, y $\omega(p)$ representa su coste. Usamos ∞ para representar el coste de los caminos a vértices que, en el momento del cómputo, no se han alcanzado.

Cuando un camino p desde 0 hasta un vértice dado i cumple $\omega(p) = \delta(i)$ entonces se dice que p es uno de los *caminos más cortos hasta el vértice* i. Cuando un camino p desde 0 hasta i pasa solo -antes de alcanzar su destino- por vértices de un conjunto W, decimos que p es un *camino especial hasta* i *con respecto a* W, y escribimos $\delta(i, W)$ para representar el coste del camino especial más corto hasta i c.r.a W. Obsérvese que $\delta(i) = \delta(i, V)$.

Básicamente, los algoritmos que vamos a ver en este trabajo mantienen, para cada vértice i, una estimación del coste de su camino más corto $c[i]$, que a su vez es una cota superior de $\delta(i)$. El coste del camino más corto para i se averigua en un bucle que termina para él cuando la estimación alcanza un mínimo, momento en el que i pasa a un conjunto de vértices llamados *frontera*, que está incluido en el conjunto de vértices *resueltos*. Ambos conjuntos se actualizan en cada iteración del bucle. Usaremos la siguiente notación para representar la menor estimación de un conjunto de vértices:

$$\overline{min}(c, W) = \begin{cases} a & si\ \exists i \in W(c[i] = a \wedge \forall j \in W \backslash \{i\}(c[j] \geq a)) \\ \infty & en\ caso\ contrario \end{cases}$$

4. Paralelización del algoritmo de Dijkstra

En este trabajo presentamos varias versiones paralelas del algoritmo de Dijkstra. Su corrección se basa en la corrección de la formulación secuencial de este algoritmo que se muestra en el Algoritmo 1. Durante su ejecución los vértices se dividen en dos conjuntos: R de *vértices resueltos* (aquellos cuyo camino más corto ya ha sido calculado) y su complementario U de *vértices no resueltos*. El algoritmo sigue un proceso voraz, y en cada iteración del bucle se ejecutan los siguientes pasos: (1) se relajan las estimaciones de los U-vértices, (2) se calcula el mínimo de estas estimaciones, y (3) todo U-vértice i con estimación mínima se pasa al conjunto R, ya que puede demostrarse que en ese momento $c[i] = \delta(i)$. Los vértices que pasan de U a R en este paso son los vértices frontera y forman el conjunto *frontera* F. Las versiones secuenciales del algoritmo de Dijkstra que vamos a presentar se centran en la noción de frontera porque luego pueden paralelizarse con mayor facilidad. Como veremos, los vértices frontera son vértices resueltos que se usan para resolver nuevos vértices.

Básicamente, el algoritmo de Dijkstra requiere tres arrays: un array de enteros c con las estimaciones de los caminos más cortos, y dos arrays de booleanos u y f para definir los conjuntos U y F, respectivamente. Obsérvese que i es un R-vértice si $\neg u[i]$. El algoritmo de Dijkstra inicializa de forma obvia estos arrays, añadiendo el vértice origen 0 a F y R, mientras que el resto de los vértices se añaden a U; la estimación inicial $c[i]$ es la peor posible, para todo vértice $i \neq 0$, y $c[0] = 0$. Hecho esto, se ejecuta un bucle que contiene las tres operaciones básicas siguientes:

1. `relax(c, f, u)` relaja la estimación de todo U-vértice, usando para ello los F-vértices.
2. `minimum(c, u)` calcula el mínimo `mssp` de las estimaciones de los U-vértices.

```
void Dijkstra_adapted_to_frontiers(c) {
  initialize(c, f, u);
  mssp=0;
  while (mssp!=INFINITY) {
    relax(c, f, u);
    mssp=minimum(c, u);
    update(c, f, u, mssp);
  }//while
}

void initialize(c, f, u) {
  forall vértice i {
    c[i]=INFINITY; f[i]=false; u[i]=true;
  }//for
  c[0]=0; f[0]=true; u[0]=false;
}
```

Algoritmo 1. *Algoritmo de Dijkstra adaptado a fronteras*

3. `update(c, f, u, mssp)` actualiza el conjunto de U-vértices, eliminando aquellos cuya estimación sea `mssp`, que son los que pasan a formar el nuevo conjunto F.

4.1. Corrección del Algoritmo 1

El Algoritmo 1 es un esqueleto cuya corrección puede derivarse de la invariancia de una determinada propiedad.

Definición 1. *Definimos las siguientes propiedades:*

- $A1 \equiv \forall i \in R\big(c[i] = \delta(i)\big)$
- $B1 \equiv \forall i \in U\big(c[i] \geq \delta(i)\big)$
- $B2(W) \equiv \forall i \in U\big(c[i] = \delta(i, W)\big)$
- $C(W) \equiv mssp = \overline{\min}\,(c, W)$
- $D \equiv \forall i \in F(i \in R)$

A grandes rasgos, $A1$ expresa que el camino más corto desde el origen a cualquier R-vértice viene dado por la correspondiente c-componente. $B1$ expresa que c guarda una cota superior del coste del camino más corto a cada U-vértice. Obsérvese que $\delta(i) = \infty$ se verifica para cada vértice inalcanzable i, luego $B1$ implica $c[i] = \infty$ en estos casos. Por ello, si $A1$ y $B1$ son invariantes del bucle del Algoritmo 1, éste computa correctamente el coste del camino más corto a los vértices resueltos o inalcanzables. La propiedad D dice que los vértices frontera son resueltos. Por último, las propiedades $B2$ y C están parametrizadas con respecto a un conjunto de vértices, que se instanciará cuando definamos el invariante del bucle del algoritmo.

Definición 2. *Sea* Inv *la conjunción de las siguientes propiedades:* $A1,\ B1,\ B2(R \cap \bar{F}),\ C(U \cup F),\ D$.

La propiedad $B2(R \cap \bar{F})$ es esencial para poder asegurar que todo camino especial más corto c.r. a $R \cap \bar{F}$ es también un camino más corto para los F-vértices. La propiedad $C(U \cup F)$ expresa que `mssp` se calcula correctamente.

La prueba de que Inv es invariante está modularizada. Lo que se hace es especificar una serie de requisitos que las tres operaciones del Algoritmo 1 deben cumplir de forma que, si particularizamos cualquiera de ellas con una

implementación concreta que satisfaga el correspondiente requisito, el algoritmo resultante es correcto.

Requisito 1. `relax(c, f, u)` satisface la siguiente especificación pre/post:

$$\{Inv\}\ \text{relax}(c, f, u)\ \{A1 \wedge B1 \wedge B2(R) \wedge D\}$$

Requisito 2. `minimum(c, u)` satisface la siguiente especificación pre/post:

$$\{A1 \wedge B1 \wedge B2(R) \wedge D\}\text{minimum}(c, u)\{A1 \wedge B1 \wedge B2(R) \wedge C(U) \wedge D\}$$

Requisito 3. `update(c, f, u, mssp)` satisface la siguiente especificación pre/post:

$$\{A1 \wedge B1 \wedge B2(R) \wedge C(U) \wedge D\}\ \text{update}(c, f, u, \text{mssp})\ \{Inv\}$$

Teorema 1. *Inv permanece invariante, siempre que las implementaciones de `relax`, `minimum` y `update` satisfagan los Requisitos 1, 2 y 3.*

La terminación del bucle también se verifica, siempre que al final de cada iteración del mismo, o `mssp` pase a valer `INFINITY` o disminuya el número de vértices no resueltos.

Requisito 4. `update(c, f, u, mssp)` hace disminuir el número de vértices no resueltos, siempre que `mssp` $\neq$ `INFINITY`.

Teorema 2. *Cualquier implementación de `relax`, `minimum` y `update` que satisfaga los Requisitos 1, 2, 3 y 4 resuelve el problema SSSP.*

4.2. Versiones secuenciales del Algoritmo 1

Antes de presentar las versiones paralelas del algoritmo anterior, nos ocupamos de sus contrapartidas secuenciales. La primera posible paralelización que veremos tiene que ver con el procedimiento `relax` y lo que hace es procesar F-vértices: "para cada vértice de la frontera, visitamos sus sucesores, relajando c para aquellos que aún están sin resolver" (Algoritmo 2).

Para demostrar que esta implementación satisface el Requisito 1, necesitamos la invariancia de nuevas propiedades.

Definición 3. *Definimos las siguientes propiedades:*

- $D1 \equiv \forall i \in R \cap \overline{F}\ \forall j \in F(c[i] < c[j])$
- $D2 \equiv \forall i \in F\ \forall j \in F(c[i] = c[j])$
- $D3 \equiv \forall i \in F\ \forall j \in U(c[i] < c[j])$
- $A2(R) \equiv \forall i \in R$ *(todo camino más corto desde 0 hasta i pasa sólo por R-vértices)*

Las primeras tres propiedades expresan la relación entre

```
void relax(c, f, u) {
  forall i do {
    if (f[i]) {
      forall j sucesor de i do {
        if (u[j])
          c[j] = min(c[j], c[i]+w[i,j]);
      }//for
    }//if
  }//for
}
```

Algoritmo 2. *`relax` procesa secuencialmente F-vértices*

```
void relax(c, f, u) {
  forall i do {
    if (u[i])
      forall j predecesor de i do {
        if (f[j])
          c[i] = min(c[i], c[j]+w[j,i]);
      }//for
    }//if
  }//for
}
```

Algoritmo 3. *`relax` procesa secuencialmente U-vértices*

los c-valores de los vértices: aquellos resueltos que no están en la frontera tienen una c-componente estrictamente menor que los vértices de la frontera ($D1$), los cuales, a su vez, tienen una c-componente estrictamente menor que los vértices no resueltos ($D3$). Además, los vértices frontera tienen todos la misma c-componente ($D2$). La cuarta propiedad se usa para establecer que $B2(R)$ se cumple en la post-condición.

Extendemos los requisitos de forma que podamos demostrar la invariancia de estas nuevas propiedades.

Requisito Extendido. Dado un Requisito con especificación pre/post asociada $\{\varphi\}$ *operación* $\{\psi\}$, el correspondiente Requisito Extendido afirma:

operación satisface la siguiente especificación pre/post: $\{\varphi \wedge D1 \wedge D2 \wedge D3 \wedge A2(R)\}$ *operación* $\{\psi \wedge D1 \wedge D2 \wedge D3 \wedge A2(R)\}$

Teorema 3. *El Algoritmo 2 satisface el Requisito Extendido 1.*

Corolario 1. *El Algoritmo 2 satisface el Requisito Extendido 1, incluso eliminando la condición if que contiene, referida a vértices no resueltos.*

El principal problema a la hora de intentar hacer paralelo el Algoritmo 2 es `c[j]= min(c[j],c[i]+w(i,j))`, que podría producir inconsistencias relacionadas con la concurrencia en el caso de que dos vértices frontera `i` e `i'` accedieran al mismo vértice no resuelto `j` y se dejara finalmente el peor valor `c[i]+w(i,j)`. Por ello presentamos el Algoritmo 3, que se centra en procesar vértices no resueltos en lugar de vértices frontera: "para cada vértice no resuelto, relajamos su c-valor, visitando sus predecesores". Como veremos, este algoritmo se ajusta mejor al modelo de programación de CUDA. Obsérvese que esta versión requiere tratar con predecesores en lugar de con sucesores.

Teorema 4. *El Algoritmo 3 satisface el Requisito Extendido 1.*

En lo que se refiere a la función `minimum`, usamos el algoritmo secuencial habitual (Algoritmo 4), que cumple trivialmente el Requisito Extendido 2. Para el procedimiento `update` proponemos el Algoritmo 5: "la nueva frontera es el conjunto de vértices no resueltos cuya estimación de camino más corto es igual al mínimo de las estimaciones; los nuevos vértices frontera se marcan como resueltos".

```
int minimum(c, u) {
  mssp = INFINITY;
  forall i do {
    if (u[i])
      mssp = min(mssp, c[i]);
  }//for
  return mssp;
}
```

Algoritmo 4. *Implementación secuencial de* `minimum`

```
void update(c, f, u, mssp) {
  forall i do {
    f[i] = false;
    if (c[i] == mssp) {
      u[i] = false;
      f[i] = true;
    }//if
  }//for
}
```

Algoritmo 5. *Implementación secuencial de* `update`

Teorema 5. *El Algoritmo 5 satisface el Requisito Extendido 3.*

Es evidente que el procedimiento `update` hace disminuir el número de vértices no resueltos, siempre que *mssp* ≠ INFINITY. Por tanto, como consecuencia de los Teoremas 2, 3, 4 y 5, las dos versiones del Algoritmo 1 basadas en los Algoritmos 2, 4 y 5, y en los Algoritmos 3, 4 y 5, resuelven el problema SSSP.

4.3. Versiones paralelas del Algoritmo 1

La principal ventaja de la formulación del Algoritmo 1 es que cada una de las operaciones básicas puede ejecutarse en paralelo. En efecto, en lo que se refiere al procedimiento `relax`, el Algoritmo 2 –que procesa vértices frontera– y el Algoritmo 3– que procesa vértices no resueltos– pueden ser paralelizados en los Algoritmos 6 y 7, lanzando un hilo por cada iteración del bucle principal, esto es, lanzando un hilo por cada vértice.

Como hemos dicho, procesar vértices frontera en paralelo podría producir inconsistencias relacionadas con la concurrencia. Para evitarlas, usamos funciones atómicas en la codificación del Algoritmo 6; en particular, la instrucción atómica de CUDA `atomicMin(x,y,z)` que permite a un solo hilo almacenar el mínimo de `y` y `z` en la variable `x`.

Teorema 6. *Los Algoritmos 6 y 7 satisfacen el Requisito Extendido 1.*

Hacer una versión paralela de la función `minimum` es una tarea más difícil, ya que es de naturaleza secuencial. Afortunadamente ya se han adaptado distintos procedimientos de reducción al modelo de streams. Por ejemplo, [12] propone un *fragment shader* para maximizar una textura cuadrada, y [13] resuelve el problema *Scan* en CUDA. En este trabajo, hemos adaptado el método `reduce3` incluido en el SDK 1.1 de CUDA [15]; presentamos el algoritmo correspondiente en la siguiente sección porque el código es dependiente del lenguaje.

```
void relax(c, f, u) {
  forall i en paralelo do {
    if (f[i]) {
      forall j sucesor de i do {
        if (u[j])
          atomicMin(c[j], c[j], c[i]+w[i,j]);
      }//for
    }//if
  }//for
}
```

Algoritmo 6. `relax` *procesa F-vértices en paralelo*

```
void relax(c, f, u) {
  forall i en paralelo do {
    if (u[i]) {
      forall j predecesor de i do {
        if (f[j])
          c[i]= min(c[i], c[j]+w[j,i]);
      }//for
    }//if
  }//for
}
```

Algoritmo 7. `relax` *procesa U-vértices en paralelo*

Por último, en lo que se refiere al procedimiento `update`, paralelizamos el Algoritmo 5 lanzando un hilo por cada vértice, como se muestra en el Algoritmo 8.

Teorema 7. *El Algoritmo 8 satisface el Requisito Extendido 3.*

Como consecuencia del Teorema 2, los Algoritmos 6, 7 y 8 pueden ser integrados de forma segura en versiones paralelas del Algoritmo 1 que, por tanto, resolverán correctamente el problema SSSP.

5. Implementaciones CUDA

Nuestra representación de los grafos dirigidos está basada en listas de adyacencia implementadas estáticamente con arrays. Cada grafo está compuesto por tres arrays: e para los arcos, ω para sus pesos, y v para acceder a la lista de adyacencia de cada vértice. Concretamente, la lista de adyacencia del vértice i ocupa en e y ω las posiciones que van desde el índice $v[i]$ hasta el índice $v[i+1]-1$ (Figura 1). Para que estos límites sirvan incluso para el

```
void update(c, f, u, mssp) {
  forall i en paralelo do {
    f[i] = false;
    if (c[i] == mssp) {
      u[i] = false;
      f[i] = true;
    }//if
  }//for
}
```

Algoritmo 8. *Implementación paralela de* `update`

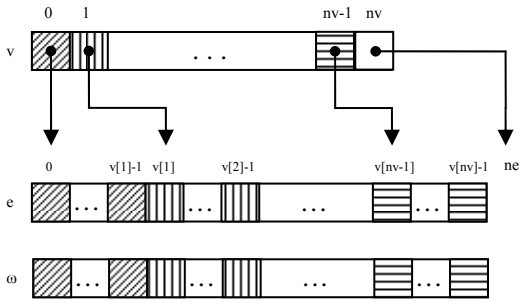


Figura 1. *Representación basada en listas de adyacencia*

último vértice, hemos añadido la componente extra $v[nv] = ne$ al final de v, donde nv y ne denotan respectivamente el número de vértices y arcos. En consecuencia, el tamaño de e y ω es ne, mientras que el de v es $nv + 1$.

Hay dos posibles formas de interpretar la información almacenada en e, ya que los vértices pertenecientes a la lista de adyacencia del vértice i pueden entenderse como predecesores o sucesores suyos. Formalmente, diremos que en la *interpretación pred*, hay un arco hacia i desde cada uno de sus vértices adyacentes, mientras que en la *interpretación suc* los arcos van desde i hacia sus adyacentes. Representar el grafo de acuerdo con la interpretación requerida es fundamental ya que el procedimiento `relax` necesita o bien la interpretación suc (Algoritmos 2 y 6) o bien la pred (Algoritmos 3 y 7), pero no ambas.

La cantidad de memoria que un grafo precisa es la suma del tamaño de los arrays `v`, `e` y `w`, esto es, $sizeof(int) * (nv + 1) + 2 * sizeof(int) * ne$ bytes. No obstante, la ejecución de los algoritmos requiere más memoria. Más concretamente, el espacio ocupado por los arrays `u`, `f` y `c`, que corresponde respectivamente a $sizeof(bool) * nv$, $sizeof(bool) * nv$, y $sizeof(int) * nv$, debe añadirse a la suma total. En cualquier caso, la cantidad de bytes resultante debe ser menor que la disponible en la tarjeta.

Para este artículo, hemos generado aleatoriamente una base de datos compuesta por grafos de grado d, es decir, la lista de adyacencia de cualquier vértice tiene d vértices. Por ello $ne = d * nv$, y la cantidad total de memoria requerida para representar el grafo y ejecutar el algoritmo es $nv * (8d + 10) + 4$ bytes. Hemos elegido $d = 7$, por lo que el grafo más grande que teóricamente podría resolverse en una tarjeta de 640 MB correspondería a $nv = (640/70) * 2^{20}$. No obstante la tarjeta necesita reservar memoria para realizar otras tareas, por lo que el tamaño mayor que hemos resuelto en la práctica ha sido de $nv = 8 * 2^{20}$. Hemos agrupado los grafos de la base de datos en ocho grupos: los grafos del n-ésimo grupo tienen $n * 2^{20}$ vértices, donde n varía de 1 a 8. El número de grafos de cada grupo es 50, lo que supone un total de 400 grafos. Como todos han sido generados usando la interpretación pred, hemos construido una representación acorde con la interpretación suc para cada uno, dando como resultado otros 400 grafos. Obsérvese que el grado de los grafos puede no conservarse tras esta operación.

5.1. Implementaciones

Hemos implementado todos los algoritmos que hemos presentado hasta ahora. Por una parte, tenemos implementaciones C para los Algoritmos secuenciales 2, 3, 4 y 5, e implementaciones CUDA para los Algoritmos paralelos 6, 7 y 8. Como todos satisfacen los requisitos extendidos de la Subsección 4.2, pueden combinarse adecuadamente para componer distintas soluciones correctas al problema SSSP. Como punto de referencia, usaremos dos implementaciones CPU: `CPU3` compuesta por los Algoritmos 3, 4 y 5, y `CPU8` formada por los Algoritmos 2, 4 y 5. De esta forma, el procedimiento `relax` procesa U-vértices en `CPU3`, y F-vértices en `CPU8`.

Los kernels de CUDA que presentaremos a continuación han sido implementados en una GeForce 8800 GTS, cuya capacidad de cómputo es 1.0. Por ello, en el procedimiento `relax` nos hemos centrado en el procesamiento de U-vértices (Algoritmo 7), ya que esta tarjeta no soporta funciones atómicas. Obsérvese que este obstáculo es frecuente en muchas de las tarjetas actuales, en particular en la popular GeForce 8800 GTX. De hecho, para poder ejecutar el Algoritmo 6 que implementa el procedimiento `relax` explorando la frontera hemos usado una GeForce 8600 GTS, que al tener capacidad de cómputo 1.1, sí las soporta.

Antes de presentar las soluciones implementadas completamente en CUDA, mencionemos que hemos probado también algunos sistemas híbridos que combinan CPU y GPU. Una característica común de todos es que ejecutan la función `minimum` en CPU por tratarse de una operación inherentemente secuencial [13][14]. Por ello, hemos probado tres implementaciones híbridas en total: `SSSP0` compuesta por los Algoritmos 3, 4 y 8, `SSSP1` por los Algoritmos 7, 4 y 5, y `SSSP2` por los Algoritmos 7, 4 y 8. En resumen, `SSSP0` ejecuta en GPU el procedimiento `update`, `SSSP1` ejecuta en GPU el procedimiento `relax` que procesa U-vértices, y `SSSP2` ejecuta en GPU ambos procedimientos, `update` y `relax`.

Para ejecutar en GPU el algoritmo completo, debemos paralelizar la función `minimum`. El Algoritmo 9 adapta el kernel `reduce3` del SDK 1.1 de CUDA para minimizar sólo las estimaciones de los U-vértices. El kernel se estructura en dos partes. En la primera, cada hilo lee dos elementos de bloques consecutivos: $i = blockIdx.x * (blockDim.x * 2) + threadIdx.x$ y $j = i + blockDim.x$. Después, escribe el mínimo de dichos elementos en memoria compartida. Obsérvese que el hilo escribe `INFINITY` si el vértice asociado ya está resuelto, con el fin de descartar estos vértices. Una vez que todos los hilos han escrito estos mínimos en memoria compartida, comienza la segunda parte. Se trata de un bucle que reduce el número de valores a la mitad en cada iteración, por lo que requiere $O(\log_2(blockDim.x))$ iteraciones para obtener el mínimo definitivo. Cada hilo activo –esto es, cada hilo cuyo número identificador `thid` en el bloque es menor que `s`, donde `s` es la distancia en la iteración actual– minimiza los valores de los índices `thid` y `thid+s`, evitando así los conflictos de bancos [13][14].

```
void minimum1(u, c, minimums) {
  forall i en paralelo do {
   thid = threadIdx.x;
   i = blockIdx.x*(2*blockDim.x)+threadIdx.x;
   j = i + blockDim.x;
   data1 = u[i] ? c[i] : INFINITY;
   data2 = u[j] ? c[j] : INFINITY;
   sdata[thid] = min(data1, data2);
   __syncthreads();
   for (s= blockDim.x/2; s>0; s>>=1) {
    if (thid<s) {
     sdata[thid]=min(sdata[thid], sdata[thid+s]);
    }// if
    __syncthreads();
   }// for
   if (thid==0) minimums[blockIdx.x] = sdata[0];
  }// forall
}
```

***Algoritmo 9**. Código CUDA para la función minimum*

Asumimos la corrección de `reduce3`, y, en consecuencia, que el Algoritmo 9 satisface el Requisito Extendido 2.

El kernel minimum1 graba el mínimo de dos bloques en el array `minimums`. Por ello, se necesitan pasadas adicionales para obtener el mínimo definitivo. Estas pasadas pueden ejecutarse en CPU o GPU. Hemos implementado otro kernel, llamado `minimum2`, para ejecutar en GPU una segunda pasada. Obviamente, este kernel no debe preocuparse de los vértices ya resueltos porque maneja los mínimos producidos por `minimum1`. Por último, los valores obtenidos tras esta segunda pasada se minimizan en CPU secuencialmente, ya que se trata de muy pocos valores. Obsérvese que éste diseño exige algunos requisitos sobre el número de elementos que pretenden minimizarse. Concretamente, este número debe ser múltiplo de $2b$ para ejecutar `minimimum1`, y múltiplo de $(2b)^2$ para ejecutar la pareja `minimimum1-minimimum2`, donde b es el número de hilos por bloque.

Nuestra primera solución implementada por completo en GPU se llama `SSSP3` e integra los Algoritmos 7, 9 y 8. Implementa el procedimiento `relax` procesando U-vértices, y aplica una única pasada de minimización mediante el kernel `minimum1`. La solución `SSSP4` la mejora ejecutando una segunda pasada de minimización mediante el kernel `minimum2`.

Hemos probado las soluciones `SSSP0`, `SSSP1`, `SSSP2`, `SSSP3` y `SSSP4` sobre nuestra base de datos, usando un Pentium IV de Intel a 3GHz y 1Gb de RAM, y una tarjeta GeForce 8800 GTS, que tiene 12 multiprocesadores y 640 MB de memoria. El código se ha desarrollado en Microsoft Visual Studio 8, usando la plantilla incluida en CUDA SDK 1.1. La Figura 2 muestra los resultados que hemos obtenido con 256 hilos por bloque. Esta configuración garantiza el 100% de la ocupación de cada multiprocesador, según indica el *CUDA GPU Occupancy Calculator v1.2* [16]. El número de vértices es $n * 2^{20}$, donde n aparece en el eje horizontal, mientras que en el vertical figuran los factores obtenidos. Estos factores se han calculado como sigue: el tiempo medio de la solución

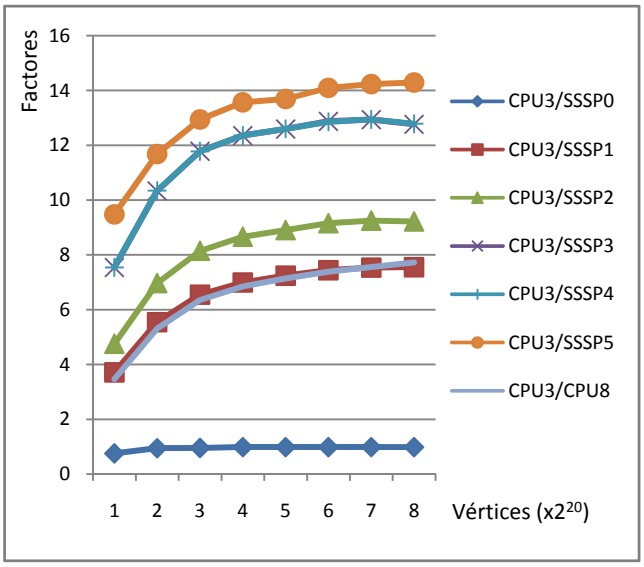


***Figura 2.** Factores en una GeForce 8800 GTS*

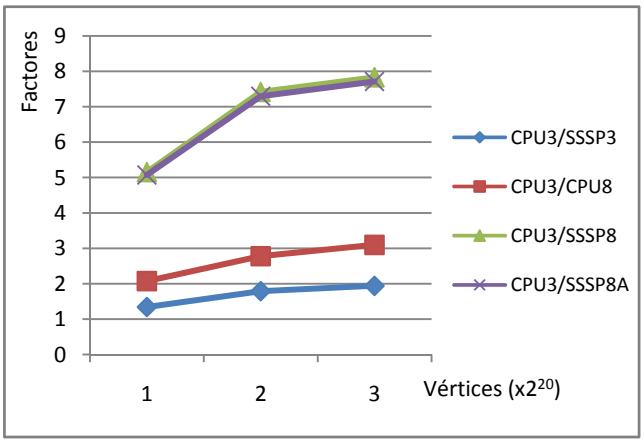


***Figura 3.** Factores en una GeForce 8600 GTS*

`CPU3` se divide entre el tiempo medio de la solución elegida. Ni la carga del grafo desde memoria secundaria, ni la inicialización de los arrays usados en los algoritmos, se incluyen en los tiempos presentados. Los resultados se analizarán en la Sección 7.

Por último, también hemos implementado la solución `SSSP8A` que se compone de los Algoritmos 6, 9 y 8. Implementa el procedimiento `relax` procesando la frontera y ejecuta en GPU una única pasada de minimización. La Figura 3 muestra los resultados que hemos obtenido para las soluciones `SSSP3` y `SSSP8A` con una GeForce 8600 GTS de capacidad de cómputo 1.1, que tiene 4 multiprocesadores y 256 MB de memoria, usando de nuevo 256 hilos por bloque. Los resultados se comentarán en la Sección 7. Por otra parte, para analizar el coste derivado de accesos simultáneos a la misma componente del array `c` dentro del kernel `relax`, hemos testeado una implementación, llamada `SSSP8`, que no usa la función `atomicMin`, sino otra función (`min`) que no es atómica. Hemos introducido esta solución simplemente por motivos de análisis estadístico, ya que, como explicaremos en la Subsección 5.3, el uso de esta función en entornos paralelos conduce a soluciones incorrectas.

5.2. Aprovechando las ventajas de CUDA

Resulta posible acelerar aún más la implementación `SSSP3` usando algunas características de CUDA. Concretamente, en esta subsección explotamos la caché de las texturas y la memoria compartida para mejorar la implementación del kernel `relax`.

Para recuperar los límites de la lista de adyacencia, el i–ésimo hilo debe leer $v[i]$ y $v[i+1]$, mientras que el $(i+1)$–ésimo lee $v[i+1]$ y $v[i+2]$. Resumiendo, ambos hilos comparten el valor $v[i+1]$, por lo que podemos usar memoria compartida para traer dicho valor una única vez. De esta forma el hilo i leería $v[i]$ de memoria global, lo escribiría en memoria compartida, y luego leería $v[i+1]$ de memoria compartida directamente. Un caso especial es el del último hilo de cada bloque, ya que tendría que traer ambos, $v[i]$ y $v[i+1]$.

Obsérvese que dos hilos pueden acceder al mismo vértice `j` del array `f`. Para acelerar las correspondientes lecturas, podemos acceder al array mediante una textura, aprovechando la caché de las texturas. Así, sería posible que hilos del mismo bloque leyeran `f[j]` de la caché en lugar de leerlo de memoria global.

El kernel resultante corresponde al Algoritmo 10. Sea `SSSP5` la solución compuesta de los Algoritmos 10, 9 y 8. La Figura 2 incluye los tiempos que hemos obtenido para `SSSP5`, usando una GeForce 8800 GTS y 256 hilos por bloque.

5.3. Una implementación errónea

A continuación explicamos los problemas que hemos encontrado en la solución al problema SSSP presentada en [10]. Los autores proponen una implementación del algoritmo de Dijkstra que explora la frontera en el procedimiento `relax`, de forma similar a nuestro Algoritmo 6. Sin embargo, en lugar de la función atómica `atomicMin`, emplean el siguiente código para relajar un vértice `j` que es sucesor de un vértice en la frontera `i`:

```
void relax(c, f, u) {
  forall i en paralelo do {
    extern __shared__ uint sdata[];
    thid = threadIdx.x + blockIdx.x + blockDim.x;
    tx = threadIdx.x;
    sdata[tx] = v[thid];
    if (tx == blockDim.x - 1)
      sdata[tx+1] = v[thid+1];
    __syncthreads();
    if (u[thid]) {
      for (i= sdata[tx]; i<sdata[tx+1]; i++) {
        pid = a[i];
        if (tex1Dfetch(tex_f, pid))
            c[thid] = min(c[thid], c[pid]+w[i]);
      }//forall
    }//if
  }//forall
}
```

Algoritmo 10. *Usando texturas y memoria compartida*

```
if (uc[j] > c[i]+w[i,j])
      uc[j] = c[i]+w[i,j];
```

donde el array `uc`, llamado el *updating cost array*, mantiene una copia de `c` antes de ejecutar `relax`. De hecho, como los autores explican, el nuevo coste no se graba en `c` directamente, sino en `uc` para evitar inconsistencias del tipo *read-after-write*. Posteriormente, se vuelca `uc` sobre `c` en el kernel `update`.

Desafortunadamente, esta técnica no es suficiente para evitar inconsistencias *write-after-write*. Concretamente, si los hilos de dos vértices de la frontera `i` e `i'` se ejecutan simultáneamente y satisfacen `uc[j]>c[i]+w[i,j]` y `uc[j]>c[i']+w[i',j]` para el mismo vértice `j`, ambos hilos harán cierta la condición del *if*. En consecuencia no habrá control sobre el valor final que se almacenará en `uc[j]`. La Figura 4 muestra un grafo específico en el que se maximiza el número de estas situaciones críticas. Hemos ejecutado su implementación en una GeForce 8800 GTS, similar a la que usan los autores en [10] (GeForce 8800 GTX), con nv=1024 y 32 hilos por bloque. De hecho, también hemos probado la versión no dirigida del grafo, ya que los autores no aclaran qué tipo de grafos resuelven. En cualquiera de los casos, obsérvese que los vértices de 1 a $nv-2$ formarán la frontera tras la primera iteración. De hecho, $c[i] = uc[i] = 1$, para $1 \le i \le nv-2$, y $c[nv-1] = uc[nv-1] =$ INFINITY, antes de la segunda iteración. Por ello, cada vértice i, con $1 \le i \le nv-2$, intentará relajar $uc[nv-1]$ con un valor diferente. Como resultado $c[nv-1]$ termina con un valor que cambia aleatoriamente de ejecución en ejecución, en lugar de calcular la solución correcta $c[nv-1] = 2$.

Debido a que los hilos de diferentes bloques no pueden sincronizarse en CUDA, la resolución de este error requiere el uso de funciones atómicas cuando el kernel `relax` procesa la frontera. Desafortunadamente estas funciones solo están disponibles a partir de la capacidad de cómputo 1.1, por lo que resolver el error en tarjetas de capacidad 1.0, como la GeForce 8800 GTS y la GTX, precisa una modificación del algoritmo más profunda. Eso es lo que hacemos al procesar vértices sin resolver en lugar de los de la frontera, en el kernel `relax`.

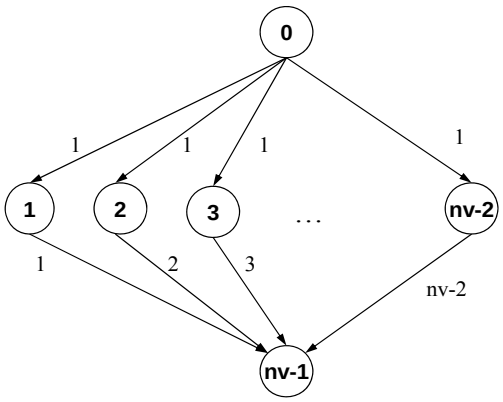


Figura 4. *Contraejemplo para [10]*

6. Grafos densos

Resulta complicado conseguir que los hilos colaboren trayendo información que puedan compartir cuando se usan listas de adyacencia. Por el contrario, cuando los grafos se representan mediante matrices de adyacencia, cada hilo debe visitar todos los elementos de una columna o de una fila, un patrón de acceso que puede aprovechar las ventajas de la memoria compartida a la hora de traer los elementos de los arrays `f`, `c` y `u` desde memoria global. Otra ventaja de esta representación consiste en que la matriz permite el acceso a los predecesores o a los sucesores del vértice `i`, sin más que explorar la columna o la fila `i`-ésima respectivamente.

El tamaño de la matriz es mayor, por lo que el número de vértices que caben en la tarjeta es menor. En concreto, la matriz ocupa $nv^2 * sizeof(int)$ bytes. Sumando además el espacio que necesitan los arrays `f`, `c` y `u`, se precisan $4nv^2 + 6nv$ bytes en total. En consecuencia, a lo sumo caben grafos de $11 * 1024$ vértices en una tarjeta 640MB. Sin embargo, la representación matricial supone la realización del mismo número de operaciones para cualquier vértice, con independencia de su grado, lo que las hace más eficaces para resolver grafos densos. Por este motivo hemos añadido a nuestra base de datos, 550 grafos representados mediante matrices de adyacencia, que hemos agrupado en once grupos de 40 grafos: los grafos del n-ésimo grupo tienen $n * 2^{10}$ vértices, donde n varía de 1 a 11. El grado de cada nodo es $nv/5$.

Como hicimos para las listas de adyacencia, solo consideraremos dos tipos de algoritmos: uno que explora predecesores (`CPU7` y `SSSP7`) y otro que procesa sucesores (`CPU9` y `SSSP9(A)`). El algoritmo paralelo `SSSP7` es similar a `SSSP3` pero usando matrices de adyacencia. Ahora el hilo t debe procesar sus predecesores visitando la columna t–ésima. Para conseguir que los hilos colaboren, la exploración se divide en segmentos de b elementos, donde b es el número de hilos por bloque. Los arrays `f` y `c` también se dividen en segmentos de b elementos. Antes de visitar un segmento de una fila, cada hilo del bloque trae un elemento de `f` y `c` a memoria compartida. De esta forma, la información de los arrays `f` y `c` está disponible durante la exploración de los predecesores asociados a ese segmento. Cuando la exploración de un segmento termina, el siguiente se procesa idénticamente. Los resultados obtenidos para `SSSP7` se muestran en la Figura 5 y se comentan en la Sección 7.

`SSSP9(A)` es la versión de `SSSP8(A)` que usa matrices de adyacencia. Los hilos podrían colaborar como en `SSSP7`, aunque trayendo esta vez elementos de `u` y `c`. Una diferencia sustancial consiste en que la operación atómica que se requeriría (`atomicMin`) debería funcionar sobre memoria compartida, ya que las actualizaciones se hacen en ella. Es cierto que ya existen tarjetas que soportan este tipo de funciones (capacidad de cómpu Vértices (x 2^{20}) hemos decidido mantener el array `c` en memoria global porque nuestras tarjetas no las soportaban. Los resultados obtenidos para `SSSP9(A)` aparecen en la Figura 6 y se analizan en la Sección 7.

7. Conclusiones

Las GPUs se pueden usar para acelerar la solución de muchos problemas, incluyendo algunos clásicos. Sin embargo, el modelo de programación de CUDA está muy restringido en todo lo que tiene que ver con la sincronización, de manera que las implementaciones se deben diseñar de forma muy cuidadosa, dando al menos intuiciones sobre la corrección de los algoritmos empleados.

En este trabajo, hemos presentado diferentes soluciones CUDA al problema SSSP, demostrando la corrección de todas ellas. Así mismo, hemos explicado el fallo que detectamos en [10], y que básicamente se debe a inconsistencias del tipo write-after-write. Para resolver este error, hemos seguido dos aproximaciones. Por una parte, se pueden usar funciones atómicas para aquellas tarjetas con capacidad de cómputo 1.1 o superior. Por otra, hemos visto cómo invertir el habitual procedimiento `relax` de forma que se procesen vértices no resueltos en lugar de vértices frontera. Nos hemos centrado sobre todo en esta solución porque es posible ejecutarla en cualquier tarjeta que posea la tecnología CUDA. La desventaja es que procesar vértices no resueltos resulta menos eficiente, ya que los arcos de entrada a un vértice no resuelto se

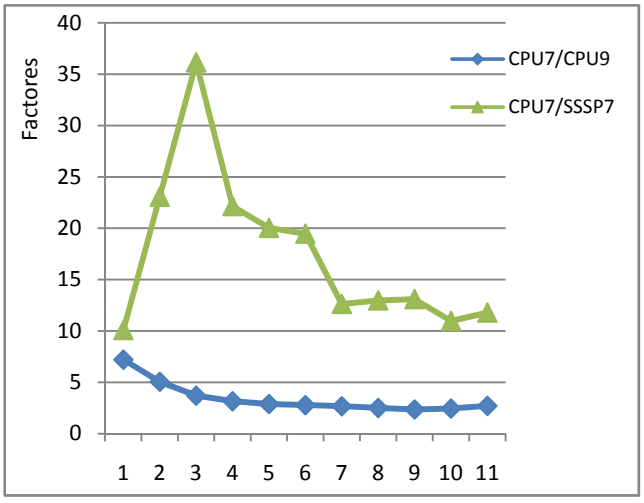


Figura 5. *Factores en una GeForce 8800 GTS*

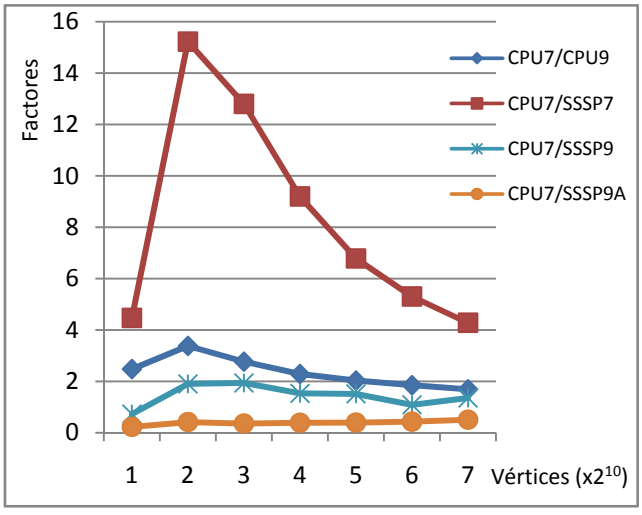


Figura 6. *Factores en una GeForce 8600 GTS*

exploran en cada iteración en la que dicho vértice sigue sin resolverse. En cambio, cuando se procesan los vértices frontera, los arcos solo se procesan una vez. Esta es la razón de que `CPU3` y `CPU7` sean menos eficientes que `CPU8` y `CPU9`, respectivamente, como se ve en todas las figuras donde aparecen.

Comparemos entonces el resto de soluciones, analizando los resultados experimentales que hemos obtenido. Para la representación basada en listas de adyacencia, la Figura 2 demuestra que las soluciones totalmente implementadas en CUDA (`SSSP3`, `SSSP4` y `SSSP5`) son más eficientes que las que solo lo están parcialmente (`SSSP0`, `SSSP1` y `SSSP2`), y ello se debe a la sobrecarga que se produce por el movimiento de datos entre CPU y GPU. Obsérvese que `SSSP0` es especialmente ineficiente ya que es incluso más lenta que `CPU3`. La Figura 2 muestra así mismo que una minimización de dos pasadas se comporta como una de pasada única, como lo refleja el hecho de que las gráficas de `SSSP3` y `SSSP4` solapen. La razón de esto se encuentra comparando el número de valores que devuelven `minimum1` y `minimum2`. Estos números son, respectivamente, $nv/(2b)$ y $nv/(2b)^2$, donde b es el número de hilos por bloque. Como nv se mueve entre $1 * 2^{20}$ y $8 * 2^{20}$, y como hemos tomado $b = 256 = 2^8$, estos números se mueven entre 2^{11} y 2^{14}, para `minimum1` y entre 2^2 y 2^5, para `minimum2`. Por tanto, el número de valores que hay que copiar de GPU en CPU, para que se calcule su mínimo en CPU, es similar tanto en `SSSP3` como en `SSSP4`; de aquí que apenas haya diferencia en el tiempo que consumen. Finalmente, la Figura 2 muestra que aprovechando los recursos de CUDA se consiguen resultados aún mejores, ya que `SSSP5`, con la que se obtiene un factor cercano a 15, es la solución más rápida.

La Figura 3 indica que el procesamiento de vértices no resueltos es más lento que el procesamiento de la frontera, incluso para las implementaciones paralelas, ya que `SSSP8A` es bastante más rápido que `SSSP3`. La segunda conclusión es que `SSSP8A` no es mucho más lento que `SSSP8`, porque el acceso simultáneo a las mismas componentes de `c` son demasiado escasas por dos motivos: el número de vértices que puede soportar una GeForce 8600 GTS y los grados elegidos son demasiado pequeños. Por último, obsérvese que `CPU8` es incluso más rápido que `SSSP3` ya que esta tarjeta solo dispone de 4 multiprocesadores.

Con respecto a las matrices de adyacencia, cuantos más vértices tenga el grafo, mayor será su grado ($= nv/5$). Por ello, las fronteras serán grandes, el número total de iteraciones pequeño, y las versiones paralelas no aprovecharán las ventajas de la tecnología CUDA. Por esta razón las curvas de las Figuras 5 y 6 decrecen según crece nv. La Figura 6 muestra que `SSSP9` es incluso más lento que `CPU9`, porque la mayoría de los hilos están activos. En consecuencia, `SSSP7` es la solución más rápida en ambas figuras: es 35 veces más rápida que `CPU7` en la Figura 5, y 15 veces en la 6. Por último, la Figura 6 ofrece más información sobre cómo afectan las operaciones atómicas al rendimiento total, pues `SSSP9A` es mucho más lenta que `SSSP9`.

Referencias

[1] Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. Num. Math. 1 (1959), 269-271.

[2] Cormen, T., Leiserson, C., Rivest, R., Stein, C. 2001. Introduction to algorithms (second edition). The MIT Press.

[3] Fredman, M. L., Tarjan, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM 34 (1987), 596-615.

[4] Meyer, U., Sanders, P. 1998. Δ-stepping: a parallel single source shortest path algorithm. In Proc. ESA'98. LNCS 1461, 393-404.

[5] Meyer, U., Sanders, P. 2003. Δ-stepping: a parallelizable shortest path algorithm. J. of Algorithms 49 (2003), 114-152.

[6] Brodal, G., Träff, J., Zaroliagis, C. 1998. A parallel priority queue with constant time operations. Journal of Parallel and Distributed Computing 49 (1998), 4-21.

[7] Madduri, K., Bader, D., Berry, J., Crobak, J. 2006. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In Proc. Workshop on Algorithm Engineering and Experiments (ALENEX'07).

[8] Di Stefano, G., Petricola, A., Zaroliagis, C. 2006. On the implementation of parallel shortest path algorithms on a supercomputer. In Proc. ISPA 2006. LNCS 4330, 406-417.

[9] http://www.nvidia.com/object/cuda_home.html#

[10] Harish, P., Narayanan, P. J. 2007. Accelerating large graph algorithms on the GPU using CUDA. In Proc. HiPC 2007. LNCS 4873, 197-208.

[11] http://www.gpgpu.org/

[12] Buck, I., Purcell, T. 2004. A toolkit for computation on GPUs. In GPU Gems, Chapter 37. Addison-Wesley.

[13] Harris, M., Sengupta, S., Owens, J. 2008. Parallel prefix sum (Scan) with CUDA. In GPU Gems 3, Chapter 39. Addison-Wesley.

[14] Harris, M. 2007. Parallel prefix sum (Scan) with CUDA. http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/scan/doc/scan.pdf

[15] http://www.nvidia.com/object/cuda_get.html

[16] http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

IMPROVING RAY TRAVERSAL BY USING SEVERAL SPECIALIZED KD-TREES

Roberto Torres[1], Pedro J. Martín[2], Antonio Gavilanes[3] and Luis F. Ayuso[4]
Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Madrid, Spain.
[1]*r.torres@fdi.ucm.es,* [2]*pjmartin@sip.ucm.es,* [3]*agav@sip.ucm.es,* [4]*lf.ayuso@pdi.ucm.es*

Keywords: Ray Tracing, Surface Area Heuristics, KD-Tree, GPU, CUDA.

Abstract: In this paper, we present several variants of the Surface Area Heuristics (SAH) to build kd-trees for specific sets of rays' directions. In order to cover the whole space of directions, several sets of directions are considered and each of them leads to a different specialized kd-tree. We call *Multi-kd-tree* to the set of these kd-trees. During rendering, each ray will traverse the kd-tree associated with the set containing its direction. In order to evaluate the efficiency of our proposal, we have implemented a *Path Tracing* and an *Ambient Occlusion* renderer on GPU with CUDA. A SAH-based kd-tree has been compared to a Multi-kd-tree and we show that all the new heuristics exhibit a better performance than SAH over usual scenes.

1 INTRODUCTION

Ray tracing algorithms cover a family of algorithms devoted to the generation of 2D images from a 3D representation of the scene. In these algorithms, rendering is carried out by shooting rays throughout the scene. The final results usually exceed in realism those obtained with the graphics pipeline algorithm. This is the reason why ray tracing is the favourite choice in the generation of photo-realistic images (Pharr and Humphreys, 2010).

A common task of every ray tracer, which is usually the most time-consuming step, is to find the nearest intersection per ray (*traversal* step). In order to accelerate this task, several data structures have been developed to organize the scene. Their advantage is that their traversal algorithms can quickly reject whole regions, avoiding many intersection tests. Examples of these structures are *uniform grids*, *kd-trees*, *octrees* and *bounding volume hierarchies (BVHs)*.

The most efficient hierarchical structures for ray tracing are built with SAH (Goldsmith and Salmon, 1987) using the greedy top-down algorithm by (MacDonald and Booth, 1990), originally presented for kd-trees, and later adapted to BVHs by (Wald, 2007). However, SAH involves assumptions about rays than can be replaced by more realistic ones to build structures with better performance during rendering (Havran and Bittner, 1999; Hunt and Mark, 2008; Fabianowski et al., 2009; Bittner and Havran, 2009).

On the other hand, GPUs are massively-parallel devices that have been used to implement ray tracers, typically binding each thread to a ray during traversal. However, a thread can stall others in the underlying SIMT architecture, mainly due to global memory readings and runtime divergences. This fact has led the design of effective GPU-based ray traversal. The first proposals (Günther et al., 2007; Popov et al., 2007), which were based on *ray packets* as traversal units, were discarded by (Aila and Laine, 2009) because many rays were forced to traverse regions of the scene they did not intersect. Nevertheless, an appropriate arrangement of rays in the device can exploit coalesced readings and cache hits of modern hardware. Therefore, recent trends use data-parallel primitives to rearrange rays in the device either at the beginning (Garanzha and Loop, 2010) or repeatedly during the traversal (Torres et al., 2011). The aim is to get a trade-off between the overload due to the rearrangement of rays and the increase of performance.

The main contribution of this paper is the development of new heuristics from a mathematical formulation of the original SAH. These heuristics specialize SAH for different sets of ray directions by restricting their domain or by assuming non-uniform probabilities. In order to cover the whole space of directions, several sets are used and a kd-tree is built for each of them. The set of these kd-trees is called a *Multi-kd-tree*. We have tested our heuristics using two ray tracing algorithms implemented with CUDA:

Path Tracing and *Ambient Occlusion*. Before traversing, secondary rays are classified and arranged on the device according to the Multi-kd-tree components. In both renderers, Multi-kd-trees exhibit better behaviour than a single SAH-based kd-tree over usual scenes, concerning traversal steps and runtime performance.

2 RELATED WORK

There is an extensive literature about acceleration structures for ray tracing. (Havran, 2000) proved that SAH-based kd-trees were very efficient concerning static scenes on CPU. Thus, subsequent work tried to move these structures from CPU to GPU. (Foley and Sugerman, 2005) presented two techniques to traverse kd-trees without a stack: *kd-tree restart* and *kd-tree backtrack*. Nevertheless, the amount of traversed nodes was greater than the one involved in the classic traversal, due to the fact that many nodes were visited several times. (Horn et al., 2007) improved kd-tree restart by using a small fixed-size stack taking advantage of the new GPU characteristics. In addition, (Popov et al., 2007) implemented a kd-tree traversal without stack on GPU by using ropes and ray packets.

As far as BVHs are concerned, (Thrane et al., 2005) was the first proposal in implementing a BVH on GPU. Afterwards, (Günther et al., 2007) designed a packet-based BVH traversal on CUDA by means of a stack that was implemented on shared memory. (Torres et al., 2009) implemented a stackless traversal on a roped BVH using packets. After that, (Aila and Laine, 2009) proved that a single-ray traversal on BVH is faster than a packet-based one due to the high memory bandwidth of GPUs. (Garanzha and Loop, 2010) developed a faster traversal by sorting the rays and breath-first traversing the BVH.

Regarding SAH, several papers have focused on improving it for specific sets of rays. (Havran and Bittner, 1999) presented several heuristics where probabilities are approximated as ratios of areas by using either orthogonal, perspective or spherical projection. Recently, (Hunt and Mark, 2008) developed a new heuristics adapted to rays in perspective space to build kd-trees by using oblique projections. (Fabianowski et al., 2009) designed a variant of SAH supposing that rays' origins are inside the scene, which is suitable for secondary and shadow rays. (Bittner and Havran, 2009) used a representative ray set to approximate the probability as the ratio of the number of intersected rays.

3 KD-TREE BASED ON SAH

A kd-tree is a binary tree responsible for organizing the objects in the scene. The volume associated with the root is the AABB (*Axis-Aligned Bounding Box*) of the whole scene and each inner node contains a plane aligned with the axes that subdivides this volume into two voxels. Thus, the volume associated with each node is the AABB that results from reducing the root's voxel with its ancestor planes. In addition, each leaf contains a list of triangles overlapping its AABB.

In order to build good kd-trees, it is essential to measure their quality. This is usually formalized by the following recursive cost function (MacDonald and Booth, 1990):

$$\begin{aligned} Cost(l) &= Cost_{tri} \cdot N_{tri}(l) \\ Cost(i) &= Cost_{plane} + P(L|i) \cdot Cost(L) \\ &\quad + P(R|i) \cdot Cost(R) \end{aligned}$$

where l is a leaf node, i is an inner node, L and R respectively denote the left and right children of i, $Cost_{tri}$ is the cost of intersecting a ray with a triangle, $N_{tri}(l)$ is the number of triangles of l, and $Cost_{plane}$ is the cost of intersecting a ray with a plane. $P(A|B)$ is the probability for any ray to intersect the AABB of node A, provided that it already intersects the AABB of node B.

The aim of the construction is to find a kd-tree with minimum cost. However, there are two values in the previous equations that have to be estimated: the probability $P(\cdot|\cdot)$ and the costs related to the children L and R.

With respect to the children's costs, trying to build all possible trees and choosing the one minimizing the cost is unfeasible in general. Therefore, children are assumed to be leaves and so, their costs are quickly computed according to the cost function. In consequence, the construction behaves as a greedy top-down algorithm that looks for the best division of an inner node into two new leaves with the lowest local cost. We follow the $O(NlogN)$ algorithm by (Wald and Havran, 2006) for the kd-tree construction.

The probability $P(A)$ can be evaluated by using geometric probability as a ratio of measures

$$P(A) = \frac{\mu(A)}{\mu(Scene)}$$

where A is the AABB of a node and *Scene* is the AABB of the whole scene —for the sake of clarity, we will identify a node with its AABB along this paper. Notice that, if A and B are AABBs inside *Scene* and $A \subseteq B$, then

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A)}{P(B)} = \frac{\mu(A)}{\mu(B)}$$

In order to specify μ, three facts are usually assumed about rays' directions (Wald and Havran, 2006):

1. All directions are equally likely, i.e. they have constant probability.
2. The origin of each ray is out of the scene.
3. The rays do not get blocked during the traversal, i.e. they finish out of the scene.

Notice that these assumptions consider directions as lines, i.e. directions ω and $-\omega$ result in the same line. So, one half of the vectors on the unit sphere are enough to cover all rays.

A particular measure μ_1 leads to the original SAH formulation as follows. Consider the function $hit_r(A)$ that returns 1 when a ray r hits the AABB A, and 0 otherwise. Under the previous assumptions, $hit_r(A)$ can be estimated as the projected area of A on any plane whose normal is the direction ω of the ray r. Thus, if we consider a set of rays, the measure is the integral over the domain of directions. As explained above, a hemisphere $\mathcal{H}$ on the unit sphere is enough to cover all directions. Mathematically, the measure μ_1 is then expressed as

$$\mu_1(A) = \int_{\omega \in \mathcal{H}} \text{proj\_orth}(A, \omega)\, d\sigma(\omega)$$

where ω is a unit ray direction, $d\sigma$ is the differential solid angle and $\text{proj\_orth}(A, \omega)$ is the area of the orthogonal projection of A on any plane whose normal is ω.

Since we work with AABBs, the latter measure can be evaluated as follows, using the hemisphere with $\omega_Z \geq 0$:

$$\mu_1(A) = \int_{\omega \in \mathcal{H}} \sum_{i \in \{X,Y,Z\}} |N_i \cdot \omega| A_i \, d\sigma(\omega)$$

$$= \int_0^{\frac{\pi}{2}} \int_0^{2\pi} (|\omega_X| \cdot A_X + |\omega_Y| \cdot A_Y + |\omega_Z| \cdot A_Z) \sin\theta \, d\phi d\theta$$

$$= 2\pi(A_X + A_Y + A_Z)$$

where A_X, A_Y and A_Z are the areas of one face of each pair of parallel faces, and $N_X = (1,0,0)$, $N_Y = (0,1,0)$ and $N_Z = (0,0,1)$ are their normals. If $SA(A)$ denotes the surface area of the AABB A, the probability $P(A|B)$ can be computed as

$$P(A|B) = \frac{\mu_1(A)}{\mu_1(B)} = \frac{2\pi(A_X + A_Y + A_Z)}{2\pi(B_X + B_Y + B_Z)} = \frac{SA(A)}{SA(B)}$$

which corresponds to the SAH formulation.

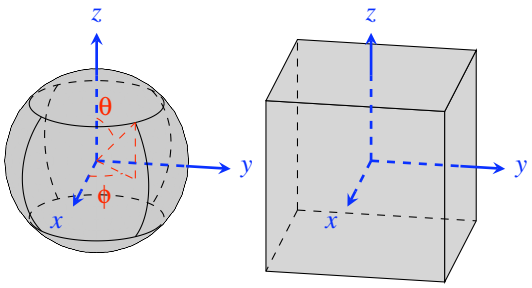


Figure 1: Distribution of the spherical patches (left) and cubic patches (right). For the sake of clarity, the six patches are shown in both figures, however, only three are considered.

4 SPECIALIZED HEURISTICS

The original SAH assumes three facts about rays (Section 3). We will define variants of SAH by changing the original assumptions about rays' directions:

1. Considering different sets of directions rather than the whole hemisphere. This leads to specialized kd-trees that result in better performance for rays whose directions belong to these sets.
2. Considering a non-uniform distribution for rays. Given a direction N, we will suppose that rays are more probable as their directions are closer to N. This results in a kd-tree specialized in the surroundings of N.

In addition, we generalize the way $hit_r(A)$ is estimated using oblique projections. Actually, we will consider orthogonal and oblique projections under the two new assumptions.

4.1 Spherical Heuristics

We relax the assumption that every ray is possible by restricting the directions to a fixed set. Nevertheless, we keep on assuming that the probability of all rays is uniform. Specifically, we split half of the direction space into three pairwise disjoint spherical patches as Figure 1 on the left shows. In that sense, the three spherical patches can be expressed as

$$SP_i = \{(\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta) \mid \theta \in \Theta_i, \phi \in \Phi_i\}$$

where $i \in \{X,Y,Z\}$, and Θ_i and Φ_i are the intervals in Table 1. The value $\theta_0 = acos(\frac{2}{3})$ has been chosen for the patches to have the same area and, therefore, the sets of directions have the same size.

As mentioned, we have two possibilities for choosing the projection. Thereby, SPHERE-ORTH and SPHERE-OBLI will respectively denote the heuristics for the orthogonal and oblique projection.

Table 1: Bounds and normalized weights for spherical and cubic heuristics. The values w_X, w_Y and w_Z are the normalized weights in percentage for the face areas A_X, A_Y and A_Z, respectively.

Patch	**Bounds (spherical coord.)**		**SPHERE-ORTH**			**SPHERE-OBLI**		
	Θ	Φ	w_X	w_Y	w_Z	w_X	w_Y	w_Z
SP_X	$[\theta_0, \pi-\theta_0]$	$[\frac{-\pi}{4}, \frac{\pi}{4}]$	55.04	22.80	22.15	53.47	23.59	22.92
SP_Y	$[\theta_0, \pi-\theta_0]$	$[\frac{\pi}{4}, \frac{3\pi}{4}]$	22.80	55.04	22.15	23.59	53.47	22.92
SP_Z	$[0, \theta_0]$	$[0, 2\pi]$	22.04	22.04	55.90	22.66	22.66	54.67
Patch	**Bounds (cartesian coord.)**			**CUBE-ORTH**			**CUBE-OBLI**	
	x	y	z	w_X	w_Y	w_Z	w_X	w_Y	w_Z
CP_X	$\{1\}$	$[-1,1]$	$[-1,1]$	51.29	24.35	24.35	50.00	25.00	25.00
CP_Y	$[-1,1]$	$\{1\}$	$[-1,1]$	24.35	51.29	24.35	25.00	50.00	25.00
CP_Z	$[-1,1]$	$[-1,1]$	$\{1\}$	24.35	24.35	51.29	25.00	25.00	50.00

In that way, each spherical patch represents a set of directions and leads to one different measure per projection type. The three measures for SPHERE-ORTH are

$$\mu_2^{(i)}(A) = \int_{\omega \in SP_i} \text{proj\_orth}(A, \omega)\, d\sigma(\omega)$$

for the patches SP_i, $i \in \{X, Y, Z\}$. For example, the probability $P(A|B)$ for patch SP_X in SPHERE-ORTH is

$$P(A|B) = \frac{\mu_2^{(X)}(A)}{\mu_2^{(X)}(B)} = \frac{w_X \cdot A_X + w_Y \cdot A_Y + w_Z \cdot A_Z}{w_X \cdot B_X + w_Y \cdot B_Y + w_Z \cdot B_Z}$$
$$= \frac{0.5504 \cdot A_X + 0.2280 \cdot A_Y + 0.2215 \cdot A_Z}{0.5504 \cdot B_X + 0.2280 \cdot B_Y + 0.2215 \cdot B_Z}$$

In general, when the integrals are solved, we obtain a weighted addition of the areas A_X, A_Y and A_Z. After that, we normalize these values by extracting their sum as a common factor. We call these normalized weights w_X, w_Y and w_Z, whose values have been included for the three spherical patches in Table 1. Notice how the area A_X has a bigger weight when considering rays with directions on the spherical patch SP_X. The use of SPHERE-ORTH leads to three different kd-trees, one for each spherical patch, i.e. the measure $\mu_2^{(i)}$ is used during the construction of the kd-tree related to SP_i.

In SPHERE-OBLI, the planes for the oblique projection must be chosen. We have tested the planes YZ for SP_X, XZ for SP_Y and XY for SP_Z. E.g., the measure for SP_Z is

$$\mu_3^{(Z)}(A) = \int_{\omega \in SP_Z} \text{proj\_obli}_{XY}(A, \omega)\, d\sigma(\omega)$$
$$= \int_{\omega \in SP_Z} \left|\frac{\omega_X}{\omega_Z}\right| A_X + \left|\frac{\omega_Y}{\omega_Z}\right| A_Y + A_Z \, d\sigma(\omega)$$

By solving the integrals and normalizing the weights, we obtain

$$P(A|B) = \frac{0.2266 \cdot A_X + 0.2266 \cdot A_Y + 0.5467 \cdot A_Z}{0.2266 \cdot B_X + 0.2266 \cdot B_Y + 0.5467 \cdot B_Z}$$

for SP_Z. See Table 1 for the normalized weights related to SP_X and SP_Y.

4.2 Cubic Heuristics

Other sets of directions can be obtained if they are taken on the surface of a cube. Similar to (Hunt and Mark, 2008), we have chosen the cube $[-1,1]^3$ as Figure 1 shows on the right. As before, directions are considered as lines, so we use three faces on the cube. They are pairwise disjoint and called cubic patches CP_X, CP_Y and CP_Z. We call CUBE-ORTH to the heuristics when the orthogonal projection is used, and CUBE-OBLI if the oblique projection is applied.

The new three measures in CUBE-ORTH are

$$\mu_4^{(i)}(A) = \int_{\omega \in CP_i} \text{proj\_orth}\left(A, \frac{\omega}{|\omega|}\right) dA(\omega)$$

for $i \in \{X, Y, Z\}$. Notice the normalization of the vector ω unlike the spherical heuristics. For example, the measure for CP_Z is

$$\mu_4^{(Z)}(A) = \int_{-1}^{1}\int_{-1}^{1} \text{proj\_orth}\left(A, \frac{(x, y, 1)}{\sqrt{x^2+y^2+1}}\right) dxdy$$

By solving and normalizing, the probability for CP_Z is

$$P(A|B) = \frac{0.2435 \cdot A_X + 0.2435 \cdot A_Y + 0.5129 \cdot A_Z}{0.2435 \cdot B_X + 0.2435 \cdot B_Y + 0.5129 \cdot B_Z}$$

In CUBE-OBLI, the oblique projection is taken into account. Using the same projection planes used for SPHERE-OBLI, we obtain the measure for CP_Z as follows

$$\mu_5^{(Z)}(A) = \int_{\omega \in CP_Z} \text{proj\_obli}_{XY}\left(A, \frac{\omega}{|\omega|}\right) dA(\omega)$$
$$= \int_{-1}^{1}\int_{-1}^{1} |x| \cdot A_X + |y| \cdot A_Y + A_Z \, dxdy$$
$$= 4\int_{0}^{1}\int_{0}^{1} x \cdot A_X + y \cdot A_Y + A_Z \, dxdy$$

Table 2: Normalized weights in percentage for cosine heuristics, taking different values of β. We only present the case for N_X. The other cases can be obtained by suitably swapping columns.

	COS-ORTH			COS-OBLI		
β	w_X	w_Y	w_Z	w_X	w_Y	w_Z
1	43.99	28.00	28.00	33.33	33.33	33.33
2	50.00	25.00	25.00	43.99	28.00	28.00
3	54.08	22.95	22.95	50.00	25.00	25.00
4	57.14	21.42	21.42	54.08	22.95	22.95
5	59.55	20.22	20.22	57.14	21.42	21.42
10	67.01	16.49	16.49	65.90	17.04	17.04

Then

$$P(A|B) = \frac{0.25 \cdot A_X + 0.25 \cdot A_Y + 0.5 \cdot A_Z}{0.25 \cdot B_X + 0.25 \cdot B_Y + 0.5 \cdot B_Z}$$

for CP_Z. Similar expressions can be obtained for CP_X and CP_Y. Table 1 displays the values of the normalized weights for these heuristics.

4.3 Cosine Heuristics

In this heuristics, we assume that all directions are possible but all of them are not equally probable. We will suppose that directions near a given unit direction N are more likely than others. We accomplish it by multiplying the projected area related to a unit direction ω by the factor $(\omega \cdot N)^\beta$, where β is a positive real number. Again, two types of projections can be considered, resulting in two heuristics, COS-ORTH for orthogonal projections and COS-OBLI for oblique projections.

We have tested three values for the direction N, $N_X = (1,0,0)$, $N_Y = (0,1,0)$ and $N_Z = (0,0,1)$. For each of them we have integrated over the hemisphere surrounding N, that is, we have used the hemispheres with $\omega_X \geq 0$, $\omega_Y \geq 0$ and $\omega_Z \geq 0$, denoted as $\mathcal{H}_X$, $\mathcal{H}_Y$ and $\mathcal{H}_Z$, respectively. Each hemisphere leads to a different measure and it produces a specific kd-tree. Notice that domains are not pairwise disjoint for the cosine heuristics.

The measures for COS-ORTH and COS-OBLI are respectively

$$\mu_6^{(i)}(A) = \int_{\omega \in \mathcal{H}_i} (\omega \cdot N_i)^\beta \cdot \text{proj\_orth}(A, \omega)\, d\sigma(\omega)$$

$$\mu_7^{(i)}(A) = \int_{\omega \in \mathcal{H}_i} (\omega \cdot N_i)^\beta \cdot \text{proj\_obli}(A, \omega)\, d\sigma(\omega)$$

for $i = \{X, Y, Z\}$. In Table 2, we present the normalized weights for N_X, taking different values of β. The weights for N_Y and N_Z result from permuting the weights for N_X, since one rotation of $\pi/2$ radians is enough to transform $\mathcal{H}_X$ into $\mathcal{H}_Y$ or $\mathcal{H}_Z$.

5 KD-TREE SELECTION

We apply the $O(NlogN)$ top-down algorithm by (Wald and Havran, 2006) for the kd-tree construction. However, instead of using the surface area to calculate the conditional probability, we apply any of the measures above described. We call kd-tree$_n^{(i)}$ to the kd-tree built with $\mu_n^{(i)}$ (the n-th measure and the set of directions SP_i or CP_i, or the normal N_i). Since, the use of a single kd-tree for the whole scene would benefit some rays but would penalize others, we build three kd-trees (kd-tree$_n^{(X)}$, kd-tree$_n^{(Y)}$ and kd-tree$_n^{(Z)}$) in order to cover the whole direction space. We call *Multi-kd-tree* to the set of these kd-trees.

The process of traversing a Multi-kd-tree by a ray in the spherical and cubic heuristics can be summarized as follows. First of all, each ray selects the kd-tree to traverse. In the case of cubic patches, it is identical to the selection of a face in the *cube mapping* technique. In the case of spherical patches, if $|\omega_Z| \geqslant \cos(\theta_0)$ then the ray chooses kd-tree$_n^{(Z)}$, and otherwise $max(|\omega_X|, |\omega_Y|)$ is used to choose kd-tree$_n^{(X)}$ or kd-tree$_n^{(Y)}$. Once a kd-tree of the Multi-kd-tree is selected by the ray, it is subsequently traversed as usual.

For the cosine heuristics, we use the kd-trees related to normals N_X, N_Y and N_Z. Each ray chooses the kd-tree to traverse by using the selection procedure of the spherical heuristics.

6 IMPLEMENTATION DETAILS

We have implemented a Path Tracing (PT) and an Ambient Occlusion (AO) on CUDA to test the performance of a Multi-kd-tree according to the new heuristics. The scenes used in our tests are BUNNY, FAIRYFOREST, CONFROOM, SPONZA and SIBENIK (Tables 6 and 7). A roof has been added to FAIRYFOREST and a bounding box enclosing BUNNY to prevent the rays from getting away from the scene. The images generated have a resolution of 1024×1024 and every surface is diffuse.

The construction of all kd-trees is made on CPU before rendering. The time spent in the construction of each kd-tree with the new heuristics is almost the same as with SAH.

Before rendering, all the kd-trees needed are allocated together on device memory. In the *node array*, all the nodes of these kd-trees are allocated, and the nodes corresponding to the same kd-tree are contiguous. In the *reference array*, the references to triangles of every leaf are stored. The indices to the root of

Table 3: Number of triangles and memory footprint used by a SAH-based kd-tree and a Multi-kd-tree built with SPHERE-ORTH. *Num.Nodes* is the number of nodes (either inner or leaf) of the kd-trees. *Num.Ref.* is the total number of references to triangles inside the leaves. Each node requires 16 bytes and each reference 4 bytes.

		SAH			SPHERE-ORTH		
Scene	Triangles	Num.Nodes	Num.Ref.	Memory	Num.Nodes	Num.Ref.	Memory
BUNNY	69,475	536,639	343,082	9.49 MB	1,738,331	1,092,768	30.69 MB
F.FOREST	174,119	1,257,457	922,883	22.70 MB	3,983,961	2,901,640	71.85 MB
CONFROOM	282,761	1,570,225	1,433,336	29.42 MB	5,253,325	4,723,711	98.17 MB
SPONZA	67,464	436,899	367,534	8.06 MB	1,339,641	1,141,669	24.79 MB
SIBENIK	80,143	358,779	311,503	6.66 MB	1,100,537	965,394	20.47 MB

each kd-tree are stored on another array, the *header array*. Table 3 shows the number of nodes (either inners or leaves) and the memory footprint used by a SAH-based kd-tree and a Multi-kd-tree built with SPHERE-ORTH. As it can be seen, the used memory of the Multi-kd-tree is about three times the space required by a SAH-based kd-tree. The remaining heuristics exhibit similar memory requirements.

Path Tracing. This renderer considers two levels of recursion: primary rays and secondary rays. It is composed of three kernels: *RayGeneration* (*RG*), *TraversalIntersection* (*TI*) and *Shading* (*SH*). The flowchart of the CUDA kernels can be seen in Figure 2 on the left. Notice that this algorithm is an *implicit* path tracer, i.e. no shadow ray is traced from the intersection points to lights. In order to complete the final image, several *iterations* of the kernels are used, being its number externally controlled.

Kernel *RG* is devoted to generating primary rays from the camera (a pinhole camera). In each iteration, four different random samples per pixel are generated, so the total amount of rays traced in parallel is $4MRays = 4 \times 1024^2$ rays per iteration. In this kernel, each ray chooses the kd-tree to traverse as already described (Section 5).

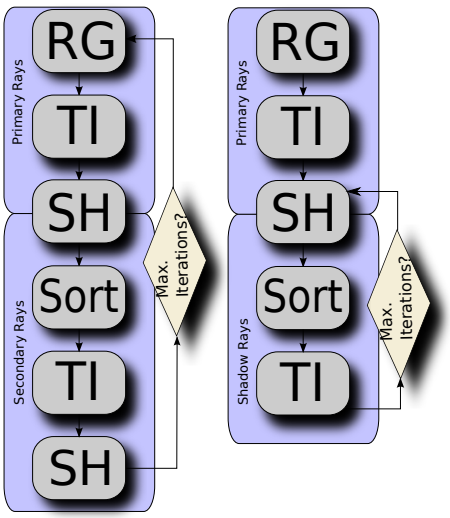


Figure 2: Flowchart of the kernels of Path Tracing (on the left) and Ambient Occlusion (on the right).

Kernel *TI* finds the nearest intersection point for each ray. This kernel is actually the algorithm *persistent while-while* by (Aila and Laine, 2009) adapted to kd-trees. At the beginning of this kernel, the header array is queried by each ray and the root of the kd-tree is retrieved to start traversing.

Kernel *SH* accumulates the color of the rays in the image buffer. If the rays are primary, then this kernel also generates the new secondary ray from each primary ray. These rays are generated on the hemisphere surface according to the cosine probability. In this kernel, and similar to *RG*, the new secondary rays choose the kd-tree to be traversed on the subsequent *TI* launching.

Ambient Occlusion. This renderer also considers two levels of recursion: primary rays and shadow rays. It is also composed of three kernels (Figure 2 on the right), which are very similar to the kernels of PT: *RayGeneration* (*RG*), *TraversalIntersection* (*TI*) and *Shading* (*SH*). In order to complete the final image, multiple iterations of the shadow rays are executed, so primary rays are only traced once at the beginning of the render. In addition, *RG* only generates one sample per pixel, so $1024^2 = 1MRays$ primary rays will be traversed in parallel. In this kernel, identically to PT, each ray selects the kd-tree to traverse.

TI has two configurations. In the first one, the kernel finds the nearest intersection point for each ray, which is suitable for primary rays. In the other, the traversal is finished as soon as an intersection point is found, which is suitable for shadow rays.

SH generates six shadow rays from each intersection point found by kernel *TI*. So $6 \times 1024^2 = 6MRays$ shadow rays will be traversed in parallel in each iteration. Each shadow ray chooses the kd-tree to be traversed, similarly to primary rays.

Ray Arrangement. Primary rays are stored on an array following the Morton code of the image pixels. In this way, contiguous rays are very likely to choose the same kd-tree to traverse. However, secondary rays

Table 4: Traversal Steps on average for Path Tracing and Ambient Occlusion. The number in parenthesis is the gain in percentage w.r.t. SAH. Bold numbers are the maximum of each row.

Path Tracing					
Primary Rays					
Scene	SAH	SPHERE-ORTH	SPHERE-OBLI	CUBE-ORTH	CUBE-OBLI
BUNNY	34.04	**30.27(11.08)**	32.30(5.11)	32.38(4.90)	32.33(5.02)
F.FOREST	48.82	**45.38(7.04)**	45.42(6.96)	45.70(6.38)	45.71(6.38)
CONFROOM	38.46	35.07(8.80)	**34.78(9.56)**	34.89(9.29)	35.01(8.96)
SPONZA	37.66	**34.52(8.33)**	34.74(7.75)	34.67(7.95)	34.97(7.13)
SIBENIK	45.03	39.39(12.51)	39.01(13.35)	**38.43(14.63)**	38.67(14.11)
Secondary Rays					
Scene	SAH	SPHERE-ORTH	SPHERE-OBLI	CUBE-ORTH	CUBE-OBLI
BUNNY	32.09	**29.85(6.99)**	30.88(3.78)	30.76(4.15)	30.75(4.18)
F.FOREST	51.51	**48.71(5.43)**	48.76(5.32)	49.11(4.65)	49.11(4.64)
CONFROOM	39.83	38.79(2.60)	38.83(2.50)	38.81(2.55)	**38.77(2.66)**
SPONZA	41.17	39.60(3.81)	39.53(3.98)	39.51(4.02)	**39.50(4.06)**
SIBENIK	48.01	46.01(4.16)	45.97(4.23)	46.02(4.13)	**45.78(4.63)**
Ambient Occlusion					
Primary Rays					
Scene	SAH	SPHERE-ORTH	SPHERE-OBLI	CUBE-ORTH	CUBE-OBLI
BUNNY	34.23	**30.46(10.99)**	32.50(5.04)	32.57(4.83)	32.53(4.96)
F.FOREST	49.03	**45.60(6.99)**	45.64(6.90)	45.92(6.33)	45.92(6.33)
CONFROOM	38.55	35.17(8.76)	**34.88(9.52)**	34.98(9.26)	35.11(8.93)
SPONZA	37.73	**34.59(8.31)**	34.81(7.73)	34.74(7.93)	35.04(7.12)
SIBENIK	47.31	41.52(12.24)	41.13(13.06)	**40.56(14.26)**	40.80(13.75)
Shadow Rays					
Scene	SAH	SPHERE-ORTH	SPHERE-OBLI	CUBE-ORTH	CUBE-OBLI
BUNNY	28.91	**26.44(8.55)**	28.07(2.90)	28.19(2.50)	28.19(2.49)
F.FOREST	42.58	**40.24(5.49)**	40.29(5.36)	40.62(4.59)	40.65(4.52)
CONFROOM	31.28	30.84(1.41)	30.82(1.46)	30.77(1.63)	**30.73(1.74)**
SPONZA	34.35	33.02(3.86)	**32.96(4.03)**	32.96(4.03)	32.90(4.20)
SIBENIK	39.55	37.93(4.10)	37.89(4.20)	37.94(4.06)	**37.82(4.36)**

are randomly generated over a hemisphere, so contiguous rays are likely to choose different kd-trees. This fact results in texture caches misses even from the beginning of *TI* since the roots of the kd-trees are very far each other. This is experimentally checked as the fact that there is fewer traversal steps w.r.t. SAH but the performance is not higher. In order to solve it, a new kernel *Sort* is added before *TI* for secondary and shadow rays and these rays are rearranged on the array. Specifically, they are sorted w.r.t. the index (to the header array) of its kd-tree. This is done on GPU using the radix sort primitive included in CUDPP 1.1.1 (Harris et al., 2010). Since at most three values are required (either one for the SAH-based kd-tree or three for the Multi-kd-tree), the sorting is carried out on the two least significant bits.

7 RESULTS

Our implementations have been tested on a NVidia GeForce 285 GTX with 1GB of DRAM on the scenes previously mentioned. The constants of the kd-tree construction are $Cost_{plane}$=1 and $Cost_{tri}$=1.

In Tables 4 and 5 we compare a single SAH-based kd-tree to a Multi-kd-tree built with our spherical and cubic heuristics. Only the kernels *TI* are measured, which are the most time-consuming according to our experiments. Specifically, traversal takes around 75%-83% of the whole rendering time. The comparison is given in traversal steps per ray on average (Table 4) and runtime performance (Table 5). A traversal step is either a plane-ray intersection or a triangle-ray intersection. The runtime performance is measured in MRays/s=1024^2 rays per second. Each scene is evaluated by positioning several cameras looking at different locations and executing several iterations per camera position.

Table 5: MRays/s for Path Tracing and Ambient Occlusion when the sorting is included (*inc.*) and not included (*n.inc.*). The number in parenthesis is the gain in percentage w.r.t. SAH. Bold numbers are the maximum of each row. For secondary and shadow rays, only columns with the sorting included (*inc.*) are considered.

Path Tracing									
Primary Rays									
Scene	SAH	SPHERE-ORTH		SPHERE-OBLI		CUBE-ORTH		CUBE-OBLI	
BUNNY	141.12	**147.45(4.29)**		144.10(2.06)		144.44(2.29)		143.68(1.77)	
F.FOREST	101.70	105.92(3.98)		105.78(3.85)		**106.04(4.09)**		105.54(3.64)	
CONFROOM	149.19	156.16(4.46)		**157.52(5.29)**		155.91(4.31)		156.78(4.84)	
SPONZA	171.75	**178.78(3.93)**		177.90(3.45)		178.41(3.73)		177.83(3.42)	
SIBENIK	143.31	155.06(7.57)		156.16(8.22)		156.66(8.52)		**156.83(8.61)**	
Secondary Rays									
Scene	SAH	SPHERE-ORTH		SPHERE-OBLI		CUBE-ORTH		CUBE-OBLI	
		n.inc.	inc.	n.inc.	inc.	n.inc.	inc.	n.inc.	inc.
BUNNY	36.29	37.14	36.12	**38.42**	**37.33**	37.72	36.67	37.33	36.30
		(2.27)	(-0.47)	**(5.53)**	**(2.78)**	(3.77)	(1.02)	(2.76)	(0.01)
F.FOREST	19.36	20.41	20.09	20.62	20.29	20.54	20.22	**20.66**	**20.33**
		(5.11)	(3.61)	(6.08)	(4.58)	(5.73)	(4.23)	**(6.25)**	**(4.75)**
CONFROOM	26.21	27.96	27.37	28.11	27.51	**28.16**	**27.57**	27.79	27.21
		(6.26)	(4.25)	(6.76)	(4.74)	**(6.94)**	**(4.93)**	(5.69)	(3.67)
SPONZA	26.14	28.47	27.83	28.52	27.91	28.45	27.84	**28.73**	**28.11**
		(8.16)	(6.08)	(8.35)	(6.34)	(8.11)	(6.10)	**(9.01)**	**(7.00)**
SIBENIK	19.66	21.72	21.37	21.76	21.40	21.71	21.35	**21.76**	**21.41**
		(9.46)	(7.95)	(9.63)	(8.12)	(9.40)	(7.90)	**(9.64)**	**(8.14)**

Ambient Occlusion									
Primary Rays									
Scene	SAH	SPHERE-ORTH		SPHERE-OBLI		CUBE-ORTH		CUBE-OBLI	
BUNNY	78.72	**81.29(3.16)**		80.36(2.03)		79.59(1.09)		79.71(1.23)	
F.FOREST	63.44	**65.09(2.53)**		65.09(2.53)		64.92(2.28)		64.85(2.18)	
CONFROOM	87.87	94.28(6.79)		**94.45(6.96)**		93.15(5.66)		94.29(6.80)	
SPONZA	112.70	**117.33(3.94)**		116.82(3.52)		117.11(3.76)		116.32(3.11)	
SIBENIK	79.41	83.08(4.40)		84.02(5.48)		83.93(5.38)		**84.55(6.07)**	
Shadow Rays									
Scene	SAH	SPHERE-ORTH		SPHERE-OBLI		CUBE-ORTH		CUBE-OBLI	
		n.inc.	inc.	n.inc.	inc.	n.inc.	inc.	n.inc.	inc.
BUNNY	**46.89**	48.00	46.33	48.28	46.59	47.04	45.44	47.19	45.58
		(2.31)	(-1.19)	(2.87)	(-0.63)	(0.32)	(-3.18)	(0.63)	(-2.87)
F.FOREST	30.80	32.02	31.21	32.26	31.45	**32.27**	**31.46**	32.23	31.43
		(3.79)	(1.30)	(4.51)	(2.07)	**(4.54)**	**(2.10)**	(4.43)	(1.99)
CONFROOM	52.80	54.88	52.57	54.72	52.43	**55.32**	**52.98**	55.11	52.78
		(3.77)	(-0.44)	(3.49)	(-0.72)	**(4.54)**	**(0.32)**	(4.18)	(-0.05)
SPONZA	47.02	50.49	48.65	**50.90**	**49.03**	50.65	48.79	50.87	49.00
		(6.87)	(3.34)	**(7.62)**	**(4.09)**	(7.15)	(3.62)	(7.55)	(4.02)
SIBENIK	37.07	39.73	38.58	39.80	38.64	**39.96**	**38.79**	39.84	38.68
		(6.68)	(3.88)	(6.83)	(4.04)	**(7.21)**	**(4.42)**	(6.95)	(4.15)

Kernel *Sort* is always launched before *TI* for secondary rays in PT and shadow rays in AO. In Table 5, the left columns of each heuristics (tagged with *n.inc.*) show the performance of kernel *TI*, not including the overload of *Sort*. On the right columns (tagged with *inc.*), the runtime of kernel *Sort* is taken into account and added to the runtime of kernel *TI*. Observe that this sorting does not affect the results in Table 4.

As it can be seen in Table 4, on average, the rays that traverse the Multi-kd-trees take less traversal steps to reach their nearest intersection points. We obtain a gain of up to 14.63% for primary rays and 6.99% for secondary ones in PT, and up to 14.26% for primary and 8.55% for shadow rays in AO. Pri-

mary rays in PT and AO are almost identical, so their results are very similar. Shadow rays in AO take fewer traversal steps than secondary rays in PT. This makes sense because the average length of shadow rays in AO is shorter than that of secondary rays in PT.

Concerning the execution model of GPUs, the traversal of different rays is not totally independent from each other. Therefore, texture cache misses and divergences can make the runtime execution different than expected. Even primary rays suffer from these stalls since the decrease in traversal steps do not agree with the improve in performance. For instance, SPHERE-ORTH takes 11.08% less traversal steps than SAH for BUNNY in PT (Table 4), but it only reaches an improvement of 4.29% in performance (Table 5). On the contrary, this clear difference does not hold for secondary rays. It is true that the sorting can entail an increase of their coherence, but secondary rays are randomly spawned and sorting only considers the kd-tree selection. Thus, reports highly depend on how these rays are concretely built during rendering.

Regarding sorting, the performance of our heuristics exceeds that of SAH when the overload due to sorting is not considered (columns *n.inc.* in Table 5). When this overload is included (columns *inc.*) our heuristics keep overcoming in most cases. The overload is more relevant in AO since shadow rays traverse fewer steps on average. Notice that scenes BUNNY and CONFROOM have the lowest average traversal steps and their results show that the overload make their runtime performance mostly slower w.r.t. SAH.

We have also compared SAH with the cosine heuristics. The settings are the same than previous heuristics. We have measured the traversal steps and the runtime performance (including *Sort*) by ranging β from 0.5 to 20 in steps of 0.5. Tables 6 and 7 show the results for COS-ORTH (blue curves) for PT and AO, respectively. The results for COS-OBLI are not depicted because they have a similar behaviour. A dashed horizontal line is added to the charts to compare this heuristics with SAH.

With respect to traversal steps in PT, it can be seen a decrease of them as β increases until it reaches a value between 2 and 3.5, for primary and secondary rays. These values of β lead to similar weights regarding the spherical and cubic heuristics. After that, the behaviour of rays becomes scene-dependant. The charts of runtime performance have an inverse behaviour, since the fewer traversal steps the rays traverse, the higher the runtime performance is.

The charts of traversal steps for primary and shadow rays have a similar shape in AO. Again, the steps traversed by shadow rays are fewer than those for secondary rays in PT due to their shorter length.

Comparing the performance charts between PT and AO, AO exhibits a better performance than PT, but the difference between COS-ORTH and SAH is larger for PT (Table 6) than for AO (Table 7). The explanation of this is the same as previous heuristics, i.e. the constant overload of sorting is more relevant for those rays with fewer traversal steps.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented six new heuristics developed from a mathematical description of the original SAH. These heuristics specialize SAH for different sets of ray directions by restricting their domain or assuming different probabilities. In order to cover the whole space of directions, several sets have been proposed and a kd-tree has been built for each of them (*Multi-kd-tree*). The traversal of a Multi-kd-tree reports fewer traversal steps and better runtime performance than a single SAH-based kd-tree over usual scenes.

However, runtime performance does not agree with the number of traversal steps, due to the execution on SIMT hardware. This fact is even more relevant for secondary or shadow rays due to their random spawning. It is necessary further research about this issue to fill the gap between traversal steps and runtime performance on parallel hardware.

A tighter division of the direction space could be realized. However, two considerations must be taken into account. First, all the information needed for the traversal has to be stored in device memory. So, a bigger amount of divisions entails more memory requirements. Second, the selection of the kd-tree to traverse has to be quick. In this work, only few comparisons are needed, which makes the selection negligible with respect to the whole traversal.

Finally, the cosine heuristics have been developed independently to the spherical and cubic heuristics. It would be interesting to analyze the behaviour of spherical or cubic patches in which rays are distributed according to the cosine heuristics.

ACKNOWLEDGEMENTS

This paper has been supported by the Spanish projects CCG10-UCM/TIC-5476 and GR35/10-A-921547. Thanks to The Stanford 3D Scanning Repository

for the BUNNY model, The Utah 3D Animation Repository for the FAIRYFOREST scene and Marko Dabrovic for the SIBENIK and SPONZA scenes.

REFERENCES

Aila, T. and Laine, S. (2009). Understanding the Efficiency of Ray Traversal on GPUs. In *High-Performance Graphics 2009*, pages 145–149.

Bittner, J. and Havran, V. (2009). RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. In *SCCG 2009*, pages 61–67, Budmerice, Slovakia.

Fabianowski, B., Flower, C., and Dingliana, J. (2009). A Cost Metric for Scene-Interior Ray Origins. In *Eurographics 2009 Short Papers*, pages 49–52.

Foley, T. and Sugerman, J. (2005). KD-Tree Acceleration Structures for a GPU Raytracer. In *Graphics Hardware 2005*, pages 15–22.

Garanzha, K. and Loop, C. (2010). Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. In *Eurographics 2010*.

Goldsmith, J. and Salmon, J. (1987). Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Application*, 7(5):14–20.

Günther, J., Popov, S., Seidel, H.-P., and Slusallek, P. (2007). Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118.

Harris, M., Owens, J. D., Sengupta, S., Tseng, S., Zhang, Y., Davidson, A., and Satish, N. (2010). CUDA Data Parallel Primitives Library (CUDPP 1.1.1). `http://code.google.com/p/cudpp/`.

Havran, V. (2000). *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Faculty of Electrical Engineering, Czech Technical University in Prague.

Havran, V. and Bittner, J. (1999). Rectilinear Trees for Preferred Ray Sets. In *SCCG 1999*, pages 171–178, Budmerice, Slovakia.

Horn, D. R., Sugerman, J., Mike, H., and Hanrahan, P. (2007). Interactive KD-Tree GPU Raytracing. In *I3D 2007*, pages 167–174.

Hunt, W. and Mark, W. R. (2008). Adaptive Acceleration Structures in Perspective Space. In *IEEE Symposium on Interactive Ray Tracing*, pages 11–17.

MacDonald, D. J. and Booth, K. S. (1990). Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(3):153–166.

Pharr, M. and Humphreys, G. (2010). *Physically Based Rendering: From Theory to Implementation (second edition)*. Morgan Kaufmann.

Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. (2007). Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum (Proceedings of Eurographics)*, 26(3):415–424.

Thrane, N., Simonsen, L. O., and Orbaek, A. P. (2005). A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Technical report, University of Aarhus.

Torres, R., Martin, P. J., and Gavilanes, A. (2011). Traversing a BVH Cut to Exploit Ray Coherence. In *GRAPP 2011*, pages 140–150.

Torres, R., Martín, P. J., and Gavilanes, A. (2009). Ray casting using a roped BVH with CUDA. In *Proc. Spring Conference on Computer Graphics*, pages 107 – 114.

Wald, I. (2007). On Fast Construction of SAH-Based Bounding Volume Hierarchies. In *Symposium on Interactive Ray Tracing 2007*, pages 33–40.

Wald, I. and Havran, V. (2006). On Building Fast KD-Trees for Ray Tracing, and on Doing That in O(NlogN). In *Symposium on Interactive Ray Tracing*, pages 61–69.

Table 6: Analysis of the COS-ORTH heuristics for Path Tracing w.r.t. traversal steps (middle column) and runtime performance (right column).

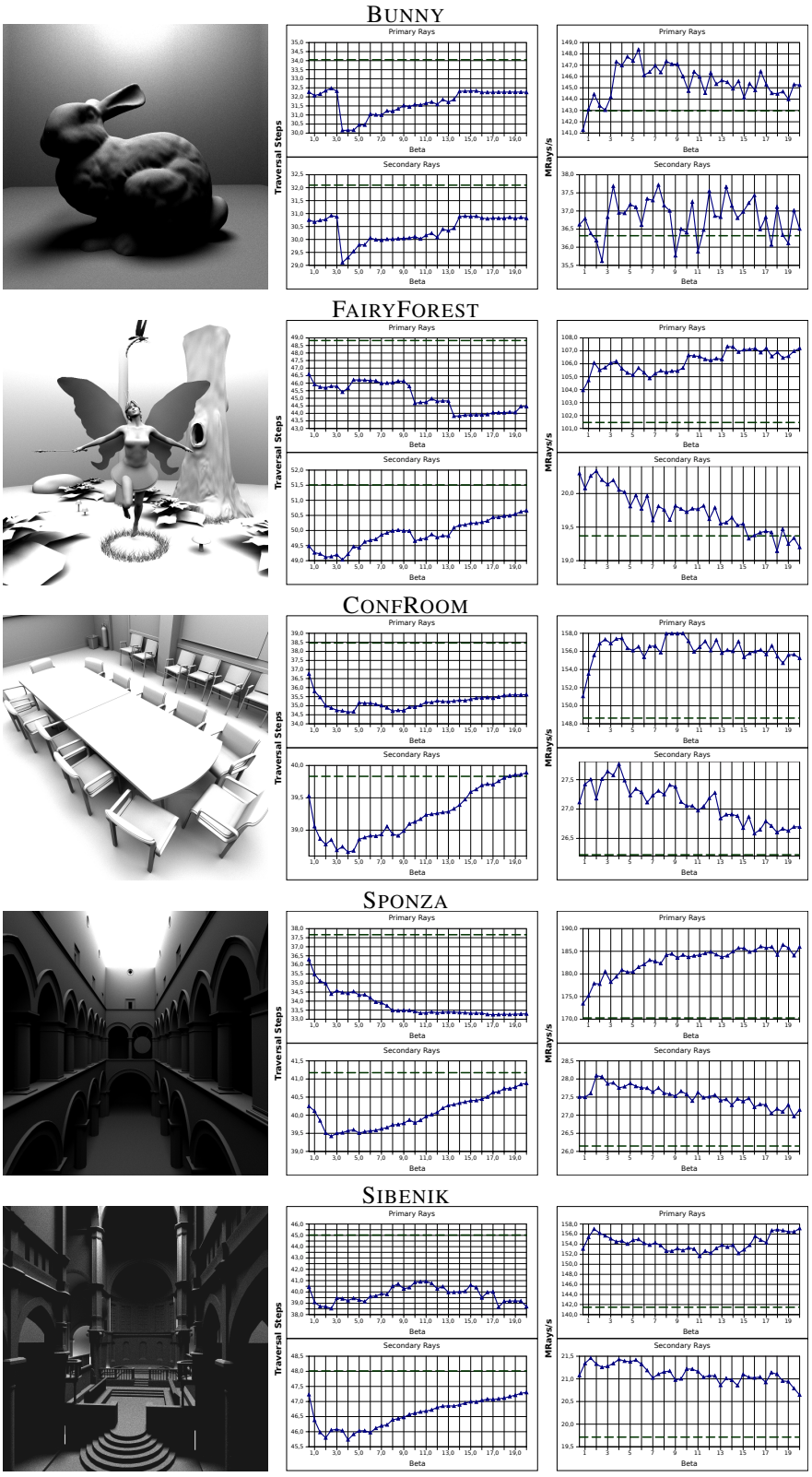
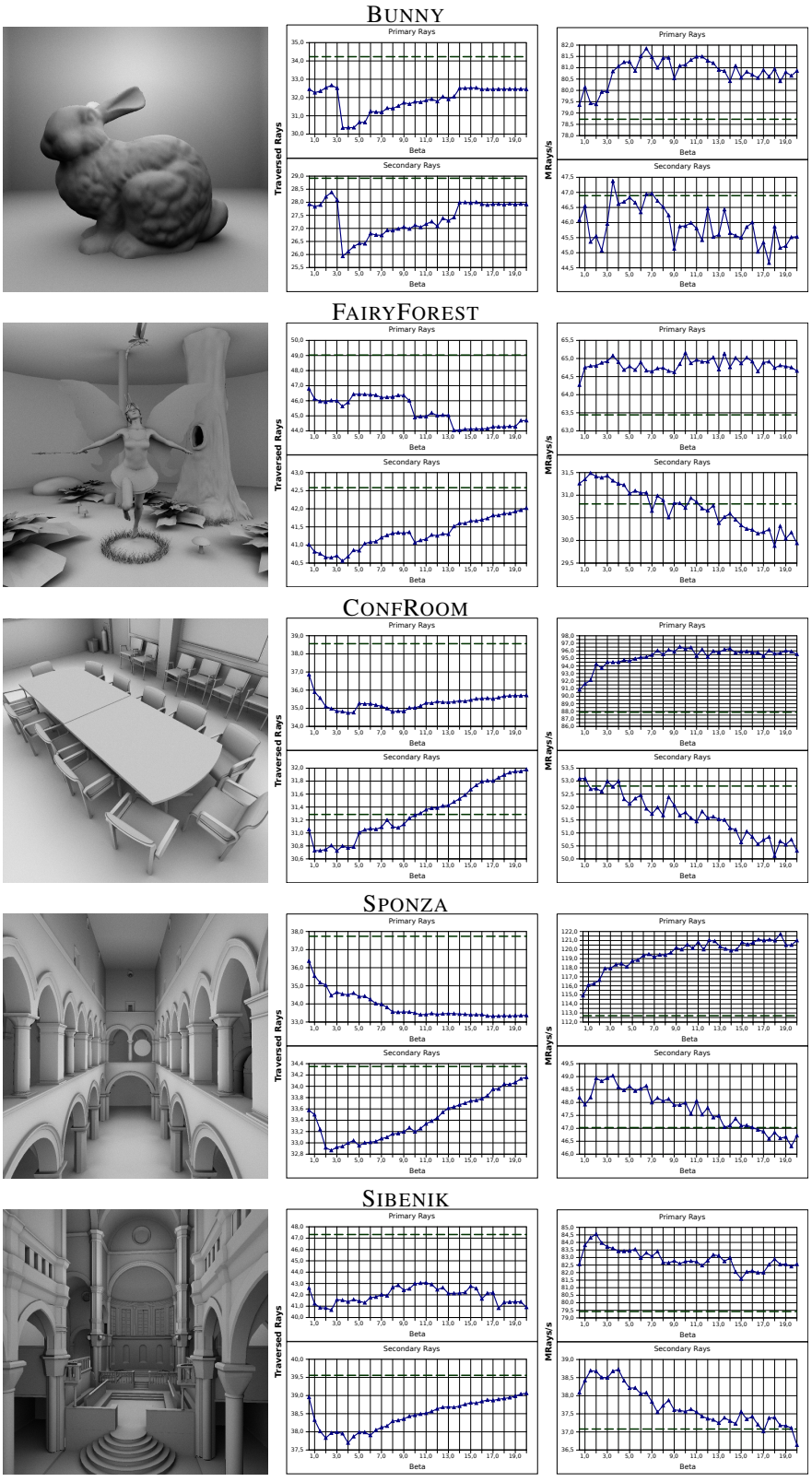


Table 7: Analysis of the COS-ORTH heuristics for Ambient Occlusion w.r.t. traversal steps (middle column) and runtime performance (right column).



Ray Tracing en CUDA usando una BVH Múltiple*

R. Torres de Alba, P. J. Martín de la Calle, A. Gavilanes Franco

Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Madrid,
{r.torres@fdi, pjmartin@sip, agav@sip}.ucm.es

Resumen

En este artículo presentamos una variante de la heurística SAH para la construcción de BVHs específicas para un conjunto determinado de rayos. Usando una BVH así construida conseguimos que los rayos pertenecientes a ese conjunto recorran la escena con mayor rapidez, aunque el resto de rayos se vean perjudicados. Para evitar esto, dividimos el espacio de direcciones en seis regiones disjuntas y construimos una BVH para cada una de ellas. Al conjunto de estas seis BVHs es a lo que llamamos BVH Múltiple. Hemos evaluado su rendimiento en dos algoritmos de ray tracing en tiempo real implementados en CUDA, usando paquetes de rayos en su recorrido.

1. Introducción

Los algoritmos de *ray tracing* comprenden una familia de algoritmos encargados de generar imágenes 2D a partir de una representación tridimensional de la escena. Una característica común de todos ellos es que exploran la escena mediante el lanzamiento de rayos. Los resultados obtenidos superan en calidad a los obtenidos por el algoritmo de *graphics pipeline*, siendo el ray tracing el algoritmo preferido en la generación de imágenes fotorrealistas [1].

Los algoritmos de ray tracing operan en tres etapas. En la primera, se generan rayos (llamados *primarios*) a partir de la posición de la cámara. En la segunda, se busca el punto de intersección más cercano de cada rayo con los objetos de la escena. En la tercera, se generan nuevos rayos (llamados *secundarios*) a partir de los puntos de intersección anteriores, que sirven para crear efectos tales como sombras, reflexiones y refracciones. La segunda y la tercera etapa se iteran hasta que no quede ningún rayo por procesar.

Dentro de esta familia de algoritmos hemos implementado dos: el *ray casting* sin sombra (RCSS) y el *ray casting* con sombra (RCCS). En el primero, se generan los rayos primarios y se calcula el punto de intersección más cercano para ellos. Posteriormente, se calcula un color para cada punto de intersección a partir del modelo de iluminación, configurando éstos la imagen final. El ray casting con sombras es una extensión del ray casting sin sombras. Una vez encontrado el punto de intersección más cercano, se genera un rayo (llamado *de sombra*) por luz que va a indicar si ese punto está visible desde dicha luz o, por el contrario, existe algún objeto que lo oculta.

De las tres etapas habituales en los algoritmos de ray tracing, la segunda es la que más tiempo consume. Usar el método de fuerza bruta y probar todos los rayos con todos los objetos haría inviable la ejecución de estos algoritmos, excepto para escenas tremendamente sencillas [2]. Para acelerar esta etapa, se han desarrollado estructuras de datos que organizan la escena y algoritmos de recorrido sobre esas estructuras que descartan regiones del espacio por las que el rayo no pasa. Ejemplos de tales estructuras son las rejillas uniformes (*uniform grids*), los *kd-trees*, los *octrees* y las *BVHs* (de *Bounding Volume Hierarchies*).

Aunque los algoritmos de ray tracing son conocidos desde hace tiempo [3], el algoritmo de *graphics pipeline* es el que ha venido dominando los gráficos interactivos. Sin embargo, trabajos recientes en CPU [4] o en GPU [5] han acelerado el ray tracing hasta conseguir su interactividad. Dos técnicas han sido fundamentales para este logro. La primera es el agrupamiento de rayos coherentes en paquetes de rayos. Llamamos rayos coherentes a aquellos que se comportan igual la mayor parte del tiempo, es decir, que prueban la intersección con los mismos nodos y con los mismos triángulos. Los paquetes de rayos permiten explotar al máximo el hardware, disminuyendo el tráfico con memoria y usando operaciones vectoriales como las SIMD. La segunda técnica es la construcción de estructuras de aceleración más eficaces mediante variantes de la heurística del área de superficie [6] (*SAH*, de *surface area heuristic*). En este trabajo pretendemos aumentar la eficiencia de las BVHs construidas con SAH para un conjunto determinado de rayos.

2. Trabajos relacionados

Las primeras aproximaciones que empleaban la GPU para el ray tracing estaban limitadas por la programabilidad de las tarjetas. El *Ray Engine* [7] fue la primera de ellas y usó la GPU para acelerar la etapa de intersección rayo-triángulo. El rendimiento estaba limitado por el tiempo requerido en enviar la geometría a la memoria de la GPU, así como por la recuperación de las intersecciones desde la GPU a la CPU.

Posteriormente, [8] implementa un ray tracing completo en GPU usando el modelo de programación en flujos de las

*Trabajo financiado por los proyectos CCG08-UCM/TIC-4252 y BSCH-UCM GR58/08-921547.

Artículo publicado en Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones (ANACAP) 2009.
ISBN 978-84-692-7320-3.

tarjetas gráficas. Al igual que en el Ray Engine, el sistema estaba muy limitado por el bajo nivel de programación de las tarjetas de la época. Esto determinó la utilización de una rejilla uniforme como estructura de aceleración debido a la simplicidad de su recorrido

Las investigaciones sobre la interactividad del ray tracing en CPU han precedido a aquellas en GPU [9], por lo que técnicas que fueron desarrolladas en un principio para CPU se han incorporado después a GPU. Éste es el caso de los paquetes de rayos [10]. Inicialmente, era una técnica para, aprovechando la coherencia de los rayos, disminuir el tráfico con memoria y aprovechar las operaciones SIMD de las CPUs. Posteriormente, fueron usadas en GPU para aprovechar los accesos fusionados a memoria por los *warps* (conjuntos de hilos que se ejecutan a la vez de manera SIMD) [11, 12].

Havran [13] demostró que los kd-trees construidos con SAH son las estructuras de aceleración más eficientes cuando se trata de escenas estáticas. Por esto, trabajos posteriores intentaron trasladar la implementación de esta estructura a GPU. Foley y Sugerman [14] presentaron dos técnicas para recorrerlos sin pila: *kd-tree restart* y *kd-tree backtrack*. Sin embargo, el número de nodos atravesados era más alto que en el recorrido clásico, debido a que muchos eran visitados más de una vez. [15] mejoró el algoritmo *kd-tree restart* usando una pila pequeña de tamaño fijo, aprovechando las nuevas opciones que las GPUs proporcionaban. Posteriormente, [11] propuso un recorrido de los kd-trees sin pila mediante hilvanes y con paquetes de rayos.

En lo que respecta a las BVHs, [16] fue el primer trabajo en implementar una BVH en GPU. Usando el enfoque clásico que equipara una GPU a un procesador de flujos, se recorría la BVH implementada según [17] sin paquetes de rayos. Posteriormente, [12] implementó en CUDA el recorrido de una BVH usando una pila en memoria compartida. En [18] encontramos una implementación en la línea de [16], pero implementada en CUDA y con paquetes de rayos.

En lo que respecta al desarrollo de estructuras de datos específicas para un determinado conjunto de rayos, se han realizado diversos trabajos. [19] presenta una estructura específica (llamada *perspective grid*) para rayos que se encuentran en el sistema de coordenadas del espacio de perspectiva. [20] desarrolló tres versiones de la SAH que usaron en la construcción de kd-trees. [21] construye un kd-tree con una heurística adaptada a rayos en coordenadas del espacio de perspectiva.

3. Jerarquías de Volúmenes Recubridores

Un volumen recubridor (*BV*, de *Bounding Volume*) es cualquier superficie tridimensional cerrada. Con mucha frecuencia, las cajas recubridoras alineadas con los ejes (*AABB*, de *Axis-Aligned Bounding Box*) son la opción preferida como BVs porque se ajustan bastante bien a los objetos que recubren y poseen un rápido algoritmo de intersección con un rayo. En este trabajo hemos elegido como BV a las AABBs.

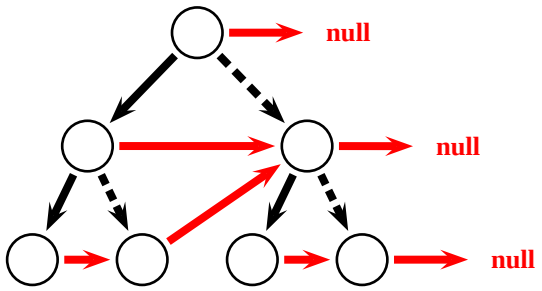


Figura 1. *Ejemplo de una BVH. Cada nodo interno posee varios hijos (flechas de color negro). A cada nodo se le ha añadido un hilván (flechas rojas) que apunta al siguiente nodo en el recorrido en preorden del árbol. Debido a la presencia de hilvanes, sólo el hijo izquierdo es necesario para el recorrido (flechas sólidas) por lo que es el único que permanecerá en la implementación.*

Una Jerarquía de Volúmenes Recubridores [22] (*BVH*) es un árbol que se encarga de organizar jerárquicamente la escena. Cada nodo del árbol tiene asociado una AABB. Las hojas poseen, además, una lista de referencias a objetos contenidos en su AABB. En una BVH se cumple que la AABB asociada a un nodo engloba a las AABBs de todos sus hijos. Esta propiedad implica que si un rayo no interseca con una AABB entonces tampoco lo hará con ninguno de sus nodos hijos. Consecuentemente, se puede descartar del recorrido un subárbol completo si el rayo no interseca con el AABB asociado a su raíz.

Según [4], una representación uniforme de la escena hace más fácil el desarrollo de implementaciones eficientes para el ray tracing. Por ello, hemos tomado la decisión de que nuestras únicas primitivas sean triángulos. Además, vamos a tratar sólo con escenas estáticas ya que la integración del dinamismo en el ray tracing no es trivial [23] y queda fuera del ámbito de este trabajo.

3.1. Recorrido para un único rayo

El recorrido de una BVH por parte de un rayo corresponde al recorrido de un árbol en preorden y con poda. En cada etapa, el algoritmo saca un nodo de la pila (inicializada con la raíz) y procede a realizar el test de intersección de la AABB de ese nodo con el rayo. Si pasa el test y es un nodo hoja, se procede a realizar la intersección con los objetos (triángulos) contenidos, almacenando la intersección más cercana. Si pasa el test y es un nodo interno, se meten todos sus hijos en la pila. En caso de que el test falle, el nodo es descartado y se procede a sacar otro de la pila. El algoritmo no puede acabar en cuanto encuentre la primera intersección, ya que (a diferencia de otras estructuras de datos como los kd-trees) los BVs pueden no ser disjuntos. Por ello, el algoritmo sólo acaba cuando la pila queda vacía, es decir, se ha recorrido el árbol entero.

El test de intersección rayo-AABB puede mejorarse si se tiene en cuenta la distancia al punto de intersección más cercano hasta ese momento. Es decir, que no es suficiente que exista intersección entre el rayo y la AABB, sino que

```
float t_min = ∞;                                    1
N_R = root;                                         2
while (N_R ≠ null) do                               3
  probar interseccion con AABB(N_R);                4
  if (existe interseccion) then                     5
    if (N_R es hoja) then                           6
      foreach (triangulo t en N_R) do               7
        probar interseccion rayo con t;             8
        actualizar t_min;                           9
      end foreach                                   10
      N_R = hilvan(N_R);                            11
    else // es un nodo interno                      12
      N_R = hijo_izq(N_R);                          13
    end if                                          14
  else  // no hay interseccion                      15
    N_R = hilvan(N_R);                              16
  end if                                            17
end while                                           18
```

Figura 2. *Algoritmo de recorrido de una BVH con un único rayo.*

ésta tiene que ser menor que la distancia mínima encontrada hasta ese momento.

El recorrido de una BVH es un proceso recursivo, por lo que es necesaria una pila. Nuestro ray tracing ha sido implementado en CUDA, por lo que el manejo de una pila debe hacerse explícito por parte del programador. Nosotros hemos optado por la solución de [16, 17, 18] que consiste en el uso de hilvanes para guiar iterativamente el proceso de recorrido, evitando así la pila. Por lo tanto, a cada nodo se le añade un hilván que apunta al siguiente nodo en el recorrido del árbol en preorden (Figura 1).

Con el uso de hilvanes, el recorrido es ahora iterativo (Figura 2). Cada rayo mantiene una referencia al siguiente nodo N_R que debe explorar, empezando por la raíz. Si N_R es un nodo interno y pasa el test de intersección rayo-AABB, entonces N_R apuntará al $hijo\_izquierdo(N_R)$. Si N_R es un nodo interno y falla el test, entonces N_R señalará al $hilvan(N_R)$. Si N_R es una hoja entonces N_R será $hilvan(N_R)$ siempre, pero el rayo realizará la intersección con los triángulos contenidos, sólo si pasa el test de intersección con la AABB de la hoja. El algoritmo acaba cuando el rayo encuentra un hilván que apunta a `null`.

3.2. Recorrido para paquetes de rayos

El uso de paquetes de rayos tiene una influencia importante en el rendimiento del ray tracing. Como veremos en la Sección 5, su uso en CUDA supone dos ventajas. Primero, tener paquetes de rayos facilita la realización de lecturas y escrituras fusionadas. Segundo, la memoria común por paquete se guarda en memoria compartida, lo que implica un ahorro de registros. Sin embargo, el uso de paquetes requiere código adicional que complica el algoritmo, pudiendo enlentecer el algoritmo general si se agrupan rayos que no son coherentes.

Sean N_P y N_R referencias a nodos de la BVH que denotan, respectivamente, el siguiente nodo del paquete y el siguiente nodo del rayo. Cada rayo posee su N_R, mientras que N_P es común a todo el paquete. Ambos valores apuntan inicialmente a la raíz de la BVH.

Llamamos rayos *activos* a aquellos que cumplen $N_R = N_P$. Los rayos activos son los únicos que van a actualizar sus referencias N_R, mientras que los *no activos* se quedan esperando (es decir, su valor N_R no cambia) hasta que se conviertan en activos de nuevo. No obstante, tanto los rayos activos como los no activos colaboran para traer información de memoria global a través de lecturas fusionadas.

```
float t_min = ∞;                                    1
N_R = root;                                         2
N_P = root;                                         3
while (N_P ≠ null) do                               4
  bool activo = (N_R == N_P);                       5
                                                    6
  // **** Actualizacion de N_R ****                 7
  colaborar para traer N_P;                         8
  barrera sincronizacion;                           9
                                                    10
  if (N_P es una hoja) then                         11
    foreach (triangulo t en N_P) do                 12
      colaborar para traer t;                       13
      barrera sincronizacion;                       14
      if (activo) then                              15
        probar interseccion rayo con t;             16
        actualizar t_min;                           17
        N_R = hilvan(N_P);                          18
      end if                                        19
    end foreach                                     20
  else    // es un nodo interno                     21
    if (activo) then                                22
      probar interseccion con AABB(N_P);            23
      if (existe interseccion) then                 24
        N_R = hijo_izq(N_P);                        25
      else                                          26
        N_R = hilvan(N_P);                          27
      end if                                        28
    end if                                          29
  end if                                            30
                                                    31
  //  **** Actualizacion de N_P ****                32
  // Algoritmo de comunicacion                      33
  shared bool algunoIzq;                            34
  algunoIzq = false;                                35
  barrera sincronizacion;                           36
  if (N_R == hijo_izq(N_P)) then                    37
    algunoIzq = true;                               38
  end if                                            39
  barrera sincronizacion;                           40
                                                    41
  // Comprobamos si alguno se ha ido                42
  // por el hijo izquierdo                          43
  if(algunoIzq) then                                44
    N_P = hijo_izq(N_P);                            45
  else                                              46
    N_P = hilvan(N_P);                              47
  end if                                            48
  barrera sincronizacion;                           49
end while                                           50
```

Figura 3. *Algoritmo de recorrido para un rayo dentro de un paquete.*

Como veremos en la Sección 5.3, por detalles de implementación, las hojas de las BVHs no poseen AABBs. Esto implica que cuando un paquete alcanza una hoja, todos los rayos activos prueban su intersección con los triángulos contenidos en ella.

El recorrido de una BVH con paquetes de rayos es el siguiente (Figura 3). Primero, todos los rayos activos actualizan su valor N_R de una manera parecida a como se hacía en el recorrido para un rayo (líneas 7-30). Después, se procede a actualizar N_P (líneas 32-49). N_P será el $hijo\_izquierdo(N_P)$ si existe algún rayo del paquete que

se haya ido por esa rama. En caso contrario (todos los rayos se han ido por el hilván), N_P será $hilvan(N_P)$. Si todos los rayos se comportan igual (esto es, el paquete es coherente la mayor parte del tiempo), entonces todos los rayos se irán por la misma dirección y N_P se actualizará apropiadamente. Si existe alguna divergencia, entonces los rayos que se han ido por el hilván permanecerán no activos hasta que sean recogidos nuevamente por el resto de rayos del paquete.

3.3. Construcción de BVHs

La eficiencia del recorrido del rayo a través de la estructura de aceleración depende en gran medida de cómo se haya construido dicha estructura (una BVH en nuestro caso). Hasta ahora los mejores resultados están basados en la heurística SAH y el algoritmo de construcción *top-down* de [24]. Aunque este algoritmo fue desarrollado inicialmente para kd-trees, fue adaptado a BVHs en [25] con un algoritmo de complejidad $\mathcal{O}(n\,log^2 n)$. Este último es el que hemos usado para la construcción de todas nuestras BVHs.

La heurística asigna un coste a cada nodo de una BVH binaria de la siguiente forma:

$$\begin{aligned} Coste_H &= Coste_{tri} \cdot N_{tri} \\ Coste_I &= 2 \cdot Coste_{BV} + P(hit_r(L)|hit_r(I)) \cdot Coste_L \\ &\quad + P(hit_r(R)|hit_r(I)) \cdot Coste_R \end{aligned}$$

donde H es un nodo hoja, I es un nodo interno, L y R son respectivamente los hijos izquierdo y derecho de I, $Coste_{tri}$ es el coste de intersecar un rayo con un triángulo, N_{tri} es el número de triángulos de la hoja H y $Coste_{BV}$ es el coste de intersecar un rayo con un BV. $P(hit_r(A)|hit_r(B))$ es la probabilidad de que el rayo r interseque el BV del nodo A sabiendo que ha intersecado con el BV del nodo B.

El algoritmo busca obtener una BVH con coste mínimo. Sin embargo, existen dos valores que no son conocidos y deben estimarse: la probabilidad condicionada y el coste de los hijos L y R. La heurística SAH nos dice que podemos aproximar la probabilidad mediante la razón entre las áreas de las superficies de los respectivos BVs:

$$P(hit_r(A)|hit_r(B)) \approx \frac{SAH(A)}{SAH(B)} = \frac{Area(A)}{Area(B)}$$

Para calcular el coste de los hijos podemos construir todos los posibles árboles para ambos hijos y quedarnos con aquellos que den el menor coste para $Coste_I$. Desgraciadamente, este algoritmo es muy costoso de evaluar, siendo sólo posible para escenas muy sencillas. Por ello, el algoritmo [25] supone que ambos hijos son nodos hojas, pudiéndose evaluar su coste como tales.

El algoritmo de construcción de BVHs es un algoritmo devorador que busca la mejor división de una hoja en dos nuevas hojas que posean un coste total menor (según la función de coste anterior). Primeramente, ordena los triángulos de la hoja según los centroides de sus AABBs en una dimensión ($k \in \{x, y, z\}$). Posteriormente, crea un plano de la forma $k = cte$ por cada centroide, que divide al nodo en dos hojas, y evalúa su coste. Este proceso se repite para cada dimensión. Una vez evaluado el coste de todos los posibles planos, se evalúa el coste de dejar el nodo como una única hoja. Si este coste es el menor de todos, entonces el nodo se deja como hoja, acabándose la recursión. Si no, el coste menor se debe a una división por un plano. Se divide entonces el nodo por ese plano y se procede recursivamente con los dos nuevos hijos creados.

4. BVH Múltiple

La heurística SAH considera que la distribución de las direcciones de los rayos es uniforme. Una BVH construida con una heurística que sólo tenga en cuenta un conjunto restringido de rayos será más eficiente para ese conjunto de rayos, tal y como se puede ver en [20]. Por ello, vamos a desarrollar una nueva heurística llamada *CSAH* (de *Cone-SAH*) para rayos cuyas direcciones formen un cono.

4.1. Heurística CSAH

Vamos a restringir el espacio de direcciones de los rayos a un cono centrado sobre uno de los ejes del sistema de coordenadas mundiales. Suponemos que dentro del cono todas las direcciones son equiprobables. Fuera del cono, las direcciones tienen probabilidad cero. Para simplificar las operaciones, vamos a restringir las direcciones del cono sólo a aquellas que están normalizadas. Además, sin pérdida de generalidad, usaremos en lo que sigue un cono centrado en el eje z (Figura 4), ya que los resultados para los restantes conos son similares.

Definimos un cono de direcciones mediante el siguiente conjunto de vectores:

$$\begin{aligned} cono(\theta_0) = \{ &(sin\theta\, cos\phi,\ sin\theta\, sin\phi,\ cos\theta) \\ &donde \quad \theta \in [0, \theta_0] \quad y \quad \phi \in [0, 2\pi] \} \end{aligned}$$

siendo el parámetro $\theta_0 \in [0, \pi/2]$ una constante que marca la amplitud del cono.

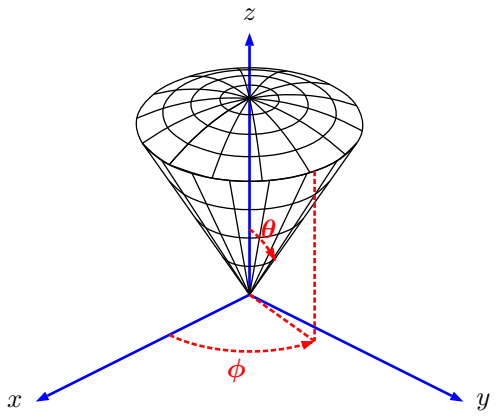


Figura 4. *Cono de direcciones centrado en el eje z.*

La probabilidad de que un rayo r interseque con una AABB A supuesto que ha intersecado con otra B puede calcularse mediante la integral de la función booleana hit sobre el cono:

$$P(hit_r(A)|hit_r(B)) \approx \frac{\int_{r \in cono(\theta_0)} hit_r(A) dr}{\int_{r \in cono(\theta_0)} hit_r(B) dr}$$

Para resolver las integrales anteriores hay que tener en cuenta que el cono sólo define un conjunto de direcciones, y que los orígenes de los rayos pueden situarse en cualquier punto del espacio. Por tanto, las anteriores integrales pueden desarrollarse de la siguiente forma:

$$\int_{r \in cono(\theta_0)} hit_r(A) dr =$$

$$\int_0^{\theta_0} \int_0^{2\pi} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} hit_r(A) do_x do_y do_z d\phi d\theta$$

Si N es la normal de una cara C de la AABB, y suponiendo fija una dirección D de un rayo r del cono (esto es, con θ y ϕ fijas), la integral triple correspondiente a la posición del ojo puede aproximarse mediante el área positiva de la proyección ortogonal de C en la dirección D. Obsérvese que el plano usado en la proyección es indiferente, ya que el área proyectada ortogonalmente es la misma para todos los planos que comparten una misma normal (D en nuestro caso). Además, el área positiva proyectada por un par de caras paralelas de una AABB corresponde al valor absoluto del área proyectada por una sola de las caras.

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} hit_r(C) do_x do_y do_z \approx abs(D \cdot N) Area(C)$$

Por tanto, podemos aproximar la probabilidad de intersecar C mediante:

$$\int_0^{\theta_0} \int_0^{2\pi} abs(D \cdot N) Area(C)\, d\phi d\theta$$

Resolviendo esta expresión para cada uno de los tres pares de caras de una AABB tenemos:

- Cara de la AABB paralela al plano YZ con normal $(1, 0, 0)$ y área $\Delta y \Delta z$:

$$\int_0^{\theta_0} \int_0^{2\pi} abs(sin\theta cos\phi)\, \Delta y \Delta z\ d\phi\, d\theta \quad =$$

$$\int_0^{\theta_0} \int_{-\pi/2}^{\pi/2} sin\theta cos\phi\ \Delta y \Delta z\ d\phi\, d\theta \quad +$$

$$\int_0^{\theta_0} \int_{\pi/2}^{3\pi/2} -sin\theta cos\phi\ \Delta y \Delta z\ d\phi\, d\theta \quad =$$

$$4(1 - cos\theta_0) \Delta y \Delta z$$

- Cara de la AABB paralela al plano XZ con normal $(0, 1, 0)$ y área $\Delta x \Delta z$:

$$\int_0^{\theta_0} \int_0^{2\pi} abs(sin\theta sin\phi)\, \Delta x \Delta z\ d\phi\, d\theta \quad =$$

$$\int_0^{\theta_0} \int_0^{\pi} sin\theta sin\phi\ \Delta x \Delta z\ d\phi\, d\theta \quad +$$

$$\int_0^{\theta_0} \int_{\pi}^{2\pi} -sin\theta sin\phi\ \Delta x \Delta z\ d\phi\, d\theta \quad =$$

$$4(1 - cos\theta_0) \Delta x \Delta z$$

- Cara de la AABB paralela al plano XY con normal $(0, 0, 1)$ y área $\Delta x \Delta y$:

$$\int_0^{\theta_0} \int_0^{2\pi} abs(cos\theta)\ \Delta x \Delta y\ d\phi\, d\theta \quad =$$

$$\int_0^{\theta_0} \int_0^{2\pi} cos\theta\ \Delta x \Delta y\ d\phi\, d\theta \quad =$$

$$2\pi sin\theta_0 \Delta x \Delta y$$

Por tanto, la probabilidad condicionada anterior, queda ahora como la razón de la suma de las tres áreas anteriores:

$$P(hit_r(A) | hit_r(A')) \approx \frac{CSAH_{cono(\theta_0)}(A)}{CSAH_{cono(\theta_0)}(A')} =$$

$$\frac{4(1 - cos\theta_0)(\Delta y \Delta z + \Delta x \Delta z) + 2\pi sin\theta_0 \Delta x \Delta y}{4(1 - cos\theta_0)(\Delta y' \Delta z' + \Delta x' \Delta z') + 2\pi sin\theta_0 \Delta x' \Delta y'}$$

Usaremos el algoritmo *top-down* anteriormente descrito para construir BVHs, pero sustituyendo la heurística SAH por la CSAH. A las BVHs construidas con SAH las llamamos SAH-BVHs, y a las construidas con CSAH las llamamos CSAH-BVHs.

4.2. Múltiples BVHs

Las CSAH-BVHs serán más eficientes para rayos que pertenezcan al cono con que fueron construidas que para el resto de rayos. Por ello, si usáramos una única CSAH-BVH para la escena completa estaríamos favoreciendo a los rayos incluidos en el cono y penalizando a los demás. En consecuencia, y dado que nuestro objetivo es permitir el movimiento libre de la cámara, necesitamos construir varias CSAH-BVHs de forma que el conjunto de conos cubra todo el espacio $\mathbb{R}^3$. En tiempo de ejecución, cada paquete de rayos selecciona la CSAH-BVH más adecuada para explorar la escena. Llamamos BVH Múltiple al conjunto de las seis CSAH-BVHs usadas para una misma escena.

En este trabajo, cada BVH Múltiple está compuesta por seis CSAH-BVHs. Cada una de ellas ha sido construida usando un cono de amplitud $\theta_0 = \pi/4$ sobre cada uno de los seis ejes $\pm x$, $\pm y$, $\pm z$.

4.3. Recorrido de una BVH Múltiple

Lo primero que debemos hacer para recorrer una BVH Múltiple es seleccionar una de sus CSAH-BVHs. Para un rayo, este procedimiento es idéntico a la selección de un plano cuando se usa *cube mapping*: la componente mayor (la componente con mayor valor absoluto) junto con el signo de esa componente determinan uno de los seis planos del cubo. Dicho plano está asociado con un eje, asociado a su vez con una CSAH-BVH. El recorrido de esa CSAH-BVH se realiza de la misma forma que en la Sección 3.1.

Para el recorrido con paquetes de rayos, vamos a suponer que el paquete es coherente. Elegimos un rayo cualquiera del paquete y seleccionamos la CSAH-BVH como en el caso de un único rayo. Una vez seleccionada, el recorrido es el mismo que el de la Sección 3.2

5. Detalles de implementación

Hemos implementado los algoritmos RCSS y RCCS en CUDA. Cada uno de ellos tiene dos versiones: una recorriendo una SAH-BVH y otra recorriendo una BVH Múltiple. Ambos han sido implementados usando el recorrido con paquetes ya que, como se verá en la Sección 5.2, son imprescindibles para realizar lecturas/escrituras fusionadas.

Hemos ejecutado nuestros algoritmos en dos tarjetas NVidia: una con *computer capability* 1.0 y otra con *computer capability* 1.3. Las capacidades de las tarjetas –sobre todo la de computer capability 1.0– afectan a la implementación de ciertas partes de la aplicación, principalmente a aquellas relacionadas con la capa de memoria y con los accesos fusionados sobre ella.

5.1. Kernels

Siguiendo el modelo de programación de flujos para ray tracing [8], RCSS consta de tres kernels y RCCS de cuatro (Figura 5). Para cada uno de ellos, disponemos de dos versiones, dependiendo de si estamos usando una SAH-BVH (a la que nos referiremos con el subíndice BVH) o una BVH Múltiple (subíndice BVHM).

Los dos primeros kernels –Genera Rayos (GR_{BVH} y GR_{BVHM}) y Traversal-Intersection (TI_{BVH} y TI_{BVHM})– son comunes tanto a RCSS como a RCCS. El kernel GR se encarga de generar los rayos primarios a partir de la información de la cámara (cámara standard de OpenGL). La versión GR_{BVHM}, además, calcula la CSAH-BVH que debe recorrer cada paquete de rayos en tiempo de ejecución. El kernel GR_{BVH} pasa el array de rayos al siguiente kernel, mientras que GR_{BVHM} envía el array de rayos más otro array de punteros en el que cada componente apunta a la raíz de la CSAH-BVH que debe recorrer cada rayo.

El segundo kernel, Traversal-Intersection (TI_{BVH} y TI_{BVHM}), se encarga de encontrar el punto de intersección más cercano para cada rayo. Cada uno envía al siguiente kernel la información de intersección por rayo, que consiste en las coordenadas del punto de intersección junto con la normal en dicho punto. La única diferencia entre TI_{BVH} y TI_{BVHM} es que el kernel TI_{BVHM} tiene que traerse de memoria global el puntero a la raíz de la CSAH-BVH antes de empezar a recorrerla.

Las dos versiones del kernel Shading (SH_{BVH} y SH_{BVHM}) son idénticas. Ambas calculan la componente de color difusa –según el modelo de *Phong*– de una luz, para cada intersección, y la acumulan en la imagen final. El manejo de varias luces se consigue lanzando este kernel una vez para cada luz.

El kernel Shading-Sombra (SHS_{BVH} y SHS_{BVHM}) calcula, al igual que SH, la componente difusa de una luz para cada intersección y, además, genera los rayos de sombra para esa luz. El kernel SHS_{BVHM}, al igual que el GR_{BVHM}, ha de calcular la CSAH-BVH que deben recorrer los rayos de sombra y enviar al siguiente kernel esa información.

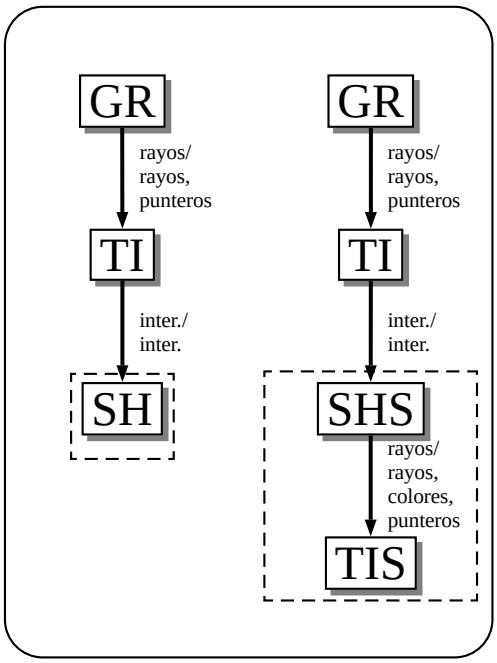


Figura 5. *Diagramas de flujos de los algoritmos. La columna de la izquierda corresponde al algoritmo RCSS y la de la derecha a RCCS. Los kernels encerrados en cajas punteadas son los kernels que se iteran para cada luz. Al lado de las flechas encontramos los nombres de los flujos de datos pasados entre kernels cuando usamos una SAH-BVH (a la izquierda de la barra) o una BVH Múltiple (a la derecha).*

El kernel Traversal-Intersection-Sombra (TIS_{BVH} y TIS_{BVHM}) es parecido al kernel TI. Sin embargo, el recorrido de un rayo de sombra es, en realidad, una consulta sobre la existencia de algún objeto que se interponga entre el punto de intersección y esa luz. Por ello, el recorrido de un rayo acaba una vez encontrada la primera intersección. En caso de que no exista intersección para ese rayo de sombra, el kernel añade el color, previamente calculado, a la imagen final. En caso contrario, no se añade nada a la imagen final, dando la apariencia de que dicho punto está en sombra para esa luz.

5.2. Paquetes de rayos

Para aprovechar el paralelismo de CUDA, lanzamos un hilo por cada rayo. Cada hilo recibe un identificador único de forma que un paquete va a estar formado por hilos con identificadores consecutivos.

A la hora de implementar los paquetes de rayos hay que tener en cuenta que el cálculo del valor siguiente de N_P requiere una comunicación entre hilos, que se lleva a cabo en memoria compartida. Esto obliga a que todos los rayos de un mismo paquete residan en el mismo bloque CUDA. Además, debe existir una sincronización a nivel de paquete (por las barreras del algoritmo de la Figura 3); como la sincronización en CUDA sólo se puede hacer a nivel de warp o de bloque, tenemos dos posibilidades a la hora de implementar un paquete: que un paquete sea un warp o un bloque. Sin embargo, por experiencias anteriores [11, 12, 18], se ha observado que resulta más eficiente que un paquete se implemente como un warp.

```
// Rayos
struct __align__(16) t_rayo1 {
   float3 origen;
   float  tmax; }
struct __align__(16) t_rayo2 {
   float3 dir;
   float  padding; }

// Punteros a CSAH-BVHs
struct punterosBVH {
   t_nodo_bvh* puntero };

// Intersecciones
struct __align__(16) t_inter1 {
   float3 punto;
   float  padding; }
struct __align__(16) t_inter2 {
   float3 normal;
   float  padding; }

// Colores
struct __align__(16) t_color {
   float3 color;
   float  padding; }
```

Figura 6. *Tipos de los elementos de los arrays de rayos, punteros a CSAH-BVHs, intersecciones y colores.*

```
// Nodos de las BVHs
struct t_nodo_bvh {
    float minAABB_X;
    float minAABB_Y;
    float minAABB_Z;
    float maxAABB_X;
    float maxAABB_Y;
    float maxAABB_Z;
    int hilvan;
    int indice; };

// Referencias a los triangulos
struct t_ref_tri {
    int indice; };
```

Figura 7. *Tipos de los elementos de los arrays de nodos y de referencias a triángulos.*

Implementar un paquete mediante un warp tiene varias consecuencias. Primero, el tamaño de todos los paquetes está fijado a 32 rayos. Para hacer que los rayos sean lo más coherente posible vamos a lanzarlos en paquetes de la forma 8×4, lo que se puede conseguir si un bloque CUDA tiene anchura 8 y altura múltiplo de 4. Segundo, ya que todos los hilos de un warp se ejecutan a la vez, no se necesitan instrucciones de sincronización.

Como ya se ha mencionado, la comunicación entre rayos del mismo paquete se realiza a través de memoria compartida. Ya que sólo es necesario saber si existe algún rayo del paquete que se haya ido por el hijo izquierdo de N_P, esto se puede implementar mediante la actualización de un flag (Figura 3, lineas 44-48)

5.3. Capa de memoria

Pensamos que los accesos a y desde memoria global son el aspecto más lento de CUDA. Por ello, es necesario realizar un esfuerzo para optimizarlos, sobre todo en tarjetas de computer capability 1.0. Nosotros hemos usado las lecturas y escrituras fusionadas como medio para acelerar estos accesos.

```
// Triangulos
struct t_triangle {
    float3 vertice0;
    float3 vertice1;
    float3 vertice2;
    float3 normal;
    float  padding[4]; };
```

Figura 8. *Tipo de los elementos del array de triángulos.*

En arquitecturas cuya computer capability es 1.0, se deben cumplir tres condiciones para que el acceso a memoria sea fusionado [26]. Primero, los datos a los que se acceden deben estar en bloques alineados de 64, 128 o 256 bytes. Segundo, cada bloque debe estar compuesto por 16 elementos de n bytes, donde n es 4 para un bloque alineado de 64 bytes, 8 para uno de 128 y 16 para uno de 256. Tercero, el i-ésimo hilo de la mitad de un warp debe acceder al i-ésimo elemento. Esto va a determinar la estructura de los arrays en memoria global. Sin embargo, las arquitecturas con computer capability 1.2 o superior tienen un patrón de acceso más flexible [26] que hace que el rendimiento no decaiga tan bruscamente si las restricciones anteriores no se cumplen.

Los arrays que mantenemos en memoria global se pueden clasificar en dos categorias: aquellos que implementan flujos de datos y aquellos que mantienen la información de la escena. Los primeros van a ser usados para pasar información entre kernels. Son el array de rayos, el array de punteros a CSAH-BVHs, el array de intersecciones y el array de colores. Para ser capaces de realizar lecturas/escrituras fusionadas es necesario que el array de rayos y el de intersecciones se dividan en dos (Figura 6).

La SAH-BVH está implementada con dos arrays: uno para la lista de nodos y otro para la lista de índices a triángulos (Figura 7). El array de nodos almacena los nodos de la BVH. Siguiendo la implementación de [17], el hijo izquierdo de cada nodo interno se encuentra en el elemento siguiente en el array, por lo que podemos ahorrarnos ese puntero. Obsérvese que tampoco es necesario guardar el puntero al hijo derecho ya que no es necesario en el recorrido de la BVH. La AABB asociada a cada nodo se guarda en los primeros 6 *floats* del elemento. El miembro `hilvan` guarda la dirección del array donde se encuentra el siguiente nodo en el recorrido. Si el nodo es una hoja, entonces el comienzo de su lista de indices a triángulos se encuentra en el campo `indice`. Para poder conocer su final tenemos que recurrir a reutilizar uno de los otros campos, en concreto nosotros hemos usado el campo `maxAABB_Z` para guardar el número de triángulos. Como consecuencia, una hoja no puede guardar su AABB. Solucionamos este inconveniente dividiendo cada hoja en dos nodos, añadiendo un nodo interno que tiene por hijo a la hoja en sí, y que sirve para guardar su AABB.

El array de triángulos (Figura 8) mantiene todos los triángulos de la escena. Cada triángulo está definido por tres vértices más su normal. No existe compartición de vértices para triángulos vecinos.

Finalmente, una BVH Múltiple está implementada con

Luces	RayCasting sin Sombras						RayCasting con Sombras					
	SAH-BVH		BVH Multiple		Ganancia		SAH-BVH		BVH Multiple		Ganancia	
	FPS	TPF	FPS	TPF	FPS	TPF	FPS	TPF	FPS	TPF	FPS	TPF
1	6.77	148.77	7.25	139.45	7.09 %	6.68 %	3.77	266.70	3.92	257.65	3.97 %	3.15 %
2	6.59	153.14	7.06	143.23	7.13 %	6.91 %	2.59	389.13	2.64	382.23	1.93 %	1.80 %
3	6.39	157.60	6.86	147.31	7.35 %	6.98 %	2.09	481.11	2.22	454.34	6.22 %	5.89 %
4	6.22	161.90	6.68	151.25	7.39 %	7.04 %	1.69	597.54	1.77	569.66	4.73 %	4.89 %
5	6.03	167.08	6.48	155.80	7.46 %	7.24 %	1.46	690.27	1.53	659.17	4.79 %	4.71 %

Tabla 1. *Resultados en FPS y en TPF (en milisegundos) de los algoritmos RCSS y RCCS sobre una GeForce 8800 GTS para la escena* Fairy Forest.

Luces	RayCasting sin Sombras						RayCasting con Sombras					
	SAH-BVH		BVH Multiple		Ganancia		SAH-BVH		BVH Multiple		Ganancia	
	FPS	TPF	FPS	TPF	FPS	TPF	FPS	TPF	FPS	TPF	FPS	TPF
1	16.67	60.95	18.54	55.24	11.21 %	10.33 %	9.80	103.18	10.43	97.37	6.42 %	5.96 %
2	15.92	63.74	17.64	57.98	10.80 %	9.93 %	6.80	148.31	7.07	143.23	3.97 %	3.54 %
3	15.32	66.20	16.90	60.40	10.31 %	9.60 %	5.48	183.93	5.89	171.72	7.48 %	7.11 %
4	14.75	68.73	16.15	63.15	9.49 %	8.83 %	4.45	226.38	4.73	213.38	6.29 %	6.09 %
5	14.16	71.54	15.51	65.67	9.53 %	8.93 %	3.88	259.58	4.09	246.45	5.41 %	5.33 %

Tabla 2. *Resultados en FPS y en TPF (en milisegundos) de los algoritmos RCSS y RCCS sobre una GeForce 280 GTX para la escena* Fairy Forest.

Algoritmo	SAH-BVH		BVH Multiple		Disminución	
	Nodos	Triángulos	Nodos	Triángulos	Nodos	Triángulos
RCSS	91.77	25.28	86.37	24.08	5.88 %	4.74 %
RCCS(1)	170.70	47.24	160.20	44.76	6.15 %	5.25 %
RCCS(2)	247.67	69.62	235.06	65.72	5.09 %	5.60 %
RCCS(3)	302.10	86.71	272.54	74.57	9.78 %	14.00 %
RCCS(4)	379.51	108.03	340.92	95.91	10.16 %	11.21 %
RCCS(5)	439.17	124.74	387.80	107.36	11.50 %	13.93 %
RCCS(10)	789.35	228.79	695.24	190.20	11.92 %	16.87 %
RCCS(20)	1512.17	439.41	1325.74	365.70	12.33 %	16.77 %
RCCS(30)	2208.40	646.57	1928.40	528.38	12.68 %	18.28 %
RCCS(40)	2937.05	862.56	2565.39	706.53	12.65 %	18.09 %
RCCS(50)	3682.51	1078.47	3213.81	884.50	12.73 %	17.99 %

Tabla 3. *Análisis del número de nodos / triángulos atravesados por los algoritmos RCSS y RCCS para la escena* Fairy Forest. *El número entre paréntesis indica el número de luces usado.*

un array cabecera de seis componentes más seis CSAH-BVHs. Cada CSAH-BVH está implementada igual que una SAH-BVH. La cabecera es un array de punteros a las CSAH-BVHs.

6. Resultados

A lo largo de este artículo, hemos ejecutado nuestros algoritmos en dos ordenadores: un Intel Core2 Quad Q9400 a 2.66GHz con una NVidia GeForce 8800 GTS (*computer capability* 1.0) y un Intel Core 2 Quad Q6600 2.4 GHz con una NVidia GeForce 280 GTX (*computer capability* 1.3).

Hemos comparado empíricamente las estructuras de datos SAH-BVH y BVH Múltiple en nuestros dos algoritmos de ray tracing en tiempo real, RCSS y RCCS. Para ello, hemos usado la escena *Fairy Forest* (Figura 9) como escena de prueba formada por 174,117 triángulos en resolución de 1024×1024. Esta escena es una buena representación de lo que puede ser una escena para un ray tracing en tiempo real. Consiste, principalmente, en un par de objetos (el hada y el árbol) altamente poligonados en medio de una amplia sala. La distribución de los triángulos no es regular y podría ser un ejemplo para el problema de *teapot in a stadium*. Las luces son luces puntuales colocadas alrededor de la escena.

El análisis se ha realizado por dos vías. Por un lado, hemos comprobado el rendimiento en tiempo real de los algoritmos en FPS (*Frames Per Second*) y en TPF (*Time Per Frame*). Ya que la estructura BVH Múltiple ha sido ideada para aplicaciones en las que la cámara se mueve libremente, hemos elegido 12 posiciones representativas para la cámara (Figura 9). En las Tablas 1 y 2 pueden verse los resultados medios para ambas estructuras en nuestros dos ordenadores. Obtenemos una ganancia en la GeForce 8800 GTS de hasta 7.46 % en FPS y de hasta 7.24 % en TPF; y en la GeForce 280 GTX de hasta 11.21 % en FPS y de hasta 10.33 % en TPF.

Vemos que, en el RCSS, a medida que aumentan las luces, la disminución en el rendimiento no es tan acusada como en el RCCS. Esto se debe a que, en el RCSS, el rendimiento decae por el reiterado lanzamiento del kernel SH y su ejecución, que consiste en la resolución de la ecuación de iluminación por cada luz. Sin embargo, en el RCCS, se ite-

ran los kernels SHS y TIS, siendo TIS un kernel muy costoso ya que ejecuta el recorrido de los paquetes de rayos de sombra por la BVH.

Por otro lado, hemos realizado un análisis del número de nodos y de triángulos que prueba cada paquete de rayos (Tabla 3). El análisis se realiza por paquetes y no por rayos independientes porque un paquete se implementa como un warp de CUDA. Esto implica que un rayo no activo consume el mismo tiempo esperando que siendo activo y probando intersecciones. Como se ve, obtenemos una disminución del número de nodos atravesados de entre el 5.09 % y el 12.73 %, y del número de triángulos atravesados de entre el 4.74 % y el 18.28 %.

7. Conclusiones y trabajo futuro

Hemos desarrollado una nueva variante de la heurística SAH pensada para rayos cuyas direcciones pertenecen a un cono. Las BVHs construidas con dicha heurística (llamadas CSAH-BVHs) son más eficientes para rayos en ese conjunto, pero menos para el resto. Para que la cámara se pueda mover libremente por la escena se crean seis CSAH-BVHs –una por cada eje de coordenadas–, abarcando así casi todo $\mathbb{R}^3$. Al conjunto de esas CSAH-BVHs es a lo que hemos llamado BVH Múltiple.

Hemos presentado también una implementación en CUDA de los algoritmos de ray casting con y sin sombras que usan tanto la estructura SAH-BVH como la BVH Múltiple. Hemos hecho una comparación del rendimiento de ambos algoritmos con ambas estructuras sobre la escena *Fairy Forest*. Hemos obtenido una ganancia en FPS de hasta 11.21 % y en TPF de hasta 10.33 %. El número de nodos atravesados por paquete disminuye a medida que el número de luces aumenta, llegando hasta 12.73 % con 50 luces. El número de triángulos también disminuye llegando hasta 17.99 % con 50 luces.

Como trabajo futuro, queda por analizar el impacto que tendrían otras divisiones del espacio, así como averiguar qué resultados se obtendrían variando el número de CSAH-BVHs usadas. Asimismo, planeamos usar esta heurística en la construcción dinámica de BVHs, utilizando el cono de la cámara como cono de direcciones.

Referencias

[1] M. Pharr, G. Humphreys, Physically Based Rendering: From Theory to Implementation, Morgan Kaufmann, 2004.

[2] T. Whitted, An improved illumination model for shaded display, Commun. ACM 23 (6) (1980) 343–349.

[3] A. Appel, Some techniques for shading machine renderings of solids, in: AFIPS 1968 Spring Joint Computer Conf., Vol. 32, 1968, pp. 37–45.

[4] I. Wald, Realtime ray tracing and interactive global illumination, Ph.D. thesis, Computer Graphics Group, Saarland University (2004).

[5] T. J. Purcell, Ray tracing on a stream processor, Ph.D. thesis, Stanford, CA, USA (2004).

[6] J. Goldsmith, J. Salmon, Automatic creation of object hierarchies for ray tracing, IEEE Comput. Graph. Appl. 7 (5) (1987) 14–20.

[7] N. A. Carr, J. D. Hall, J. C. Hart, The ray engine, in: HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, pp. 37–46.

[8] T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan, Ray tracing on programmable graphics hardware, ACM Transactions on Graphics 21 (3) (2002) 703–712.

[9] S. Parker, W. Martin, P. pike J. Sloan, P. Shirley, B. Smits, C. Hansen, Interactive ray tracing, in: Symposium on interactive 3D graphics, 1999, pp. 119–126.

[10] I. Wald, C. Benthin, M. Wagner, P. Slusallek, Interactive rendering with coherent ray tracing, in: Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001), 2001.

[11] S. Popov, J. Günther, H.-P. Seidel, P. Slusallek, Stackless kd-tree traversal for high performance GPU ray tracing, Computer Graphics Forum (Proceedings of Eurographics) 26 (3) (2007) 415–424.

[12] J. Günther, S. Popov, H.-P. Seidel, P. Slusallek, Realtime ray tracing on gpu with bvh-based packet traversal, in: Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007, 2007, pp. 113–118.

[13] V. Havran, Heuristic ray shooting algorithms, Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague (November 2000).

[14] T. Foley, J. Sugerman, Kd-tree acceleration structures for a gpu raytracer, in: HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, ACM, 2005, pp. 15–22.

[15] D. R. Horn, J. Sugerman, M. Houston, P. Hanrahan, Interactive k-d tree gpu raytracing, in: I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games, ACM, 2007, pp. 167–174.

[16] N. Thrane, L. O. Simonsen, A. P. Ørbæk, A comparison of acceleration structures for gpu assisted ray tracing, Tech. rep., University of Aarhus (2005).

[17] B. Smits, Efficiency issues for ray tracing, Journal of Graphics Tools 3 (1998) 1–14.

[18] R. Torres, P. J. Martín, A. Gavilanes, Ray casting using a roped bvh with cuda, in: Spring Conference on Computer Graphics, 2009, pp. 107 – 114.

[19] W. Hunt, W. R. Mark, Ray-specialized acceleration structures for ray tracing, in: IEEE/EG Symposium on Interactive Ray Tracing 2008, IEEE/EG, 2008, pp. 3–10.

[20] V. Havran, J. Bittner, Rectilinear bsp trees for preferred ray sets, in: Proceedings of the Spring Conference on Computer Graphics (SCCG'99).

[21] W. Hunt, W. Mark, Adaptive acceleration structures in perspective space, in: Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on, 2008, pp. 11–17.

[22] S. M. Rubin, T. Whitted, A 3-dimensional representation for fast rendering of complex scenes, in: SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques, ACM, 1980, pp. 110–116.

[23] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, P. Shirley, State of the art in ray tracing animated scenes, in: D. Schmalstieg, J. Bittner (Eds.), STAR Proceedings of Eurographics 2007, 2007, pp. 89–116.

[24] D. J. MacDonald, K. S. Booth, Heuristics for ray tracing using space subdivision, Vis. Comput. 6 (3) (1990) 153–166.

[25] I. Wald, S. Boulos, P. Shirley, Ray tracing deformable scenes using dynamic bounding volume hierarchies, ACM Transaction on Graphics 26 (1) (2007) 6.

[26] NVIDIA, Nvidia website. `www.nvidia.org`.

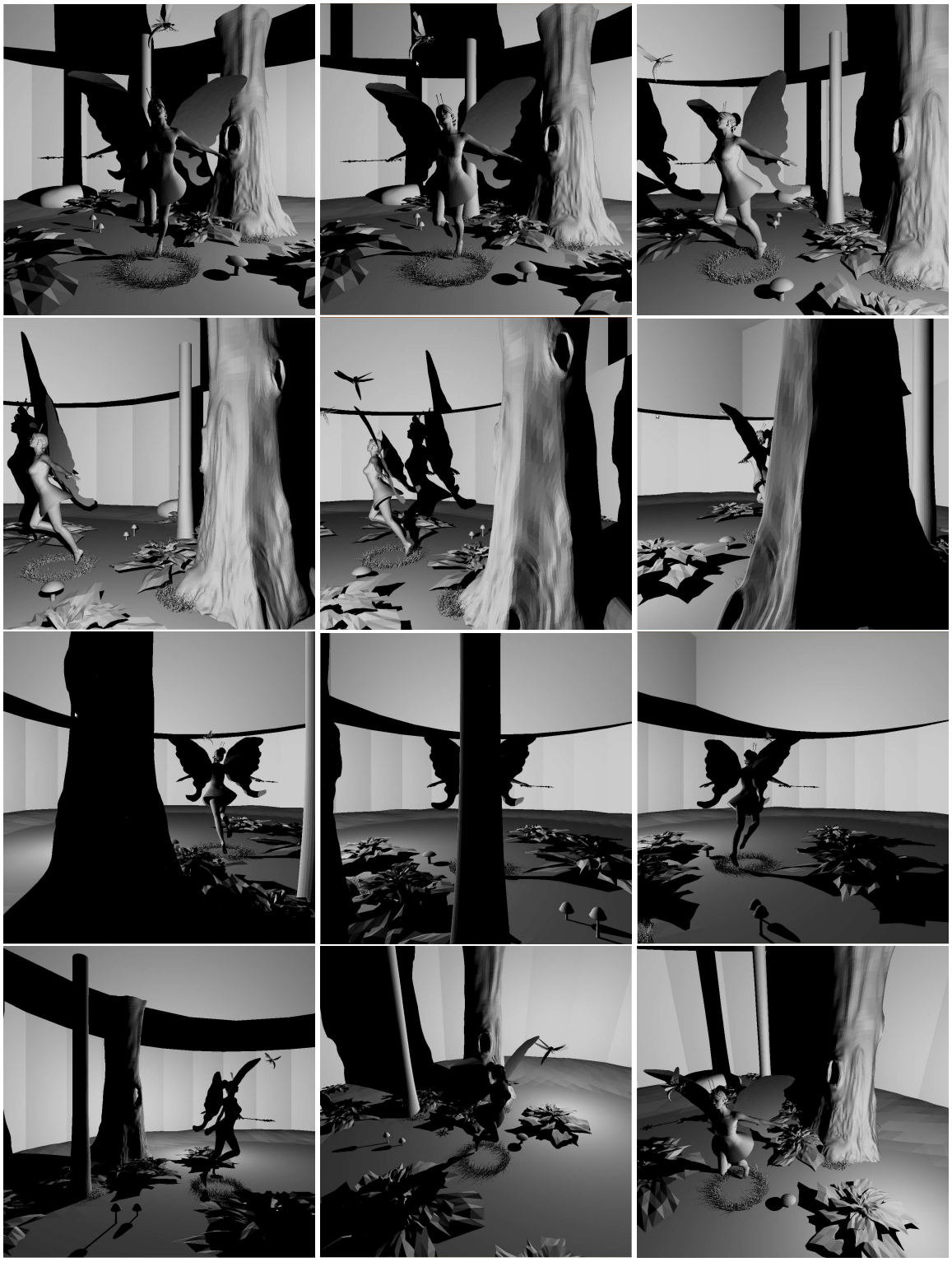


Figura 9. *Posiciones de la cámara usadas para la escena del* Fairy Forest.

Ray Casting using a Roped BVH with CUDA*

Roberto Torres† Pedro J. Martín‡ Antonio Gavilanes§
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain

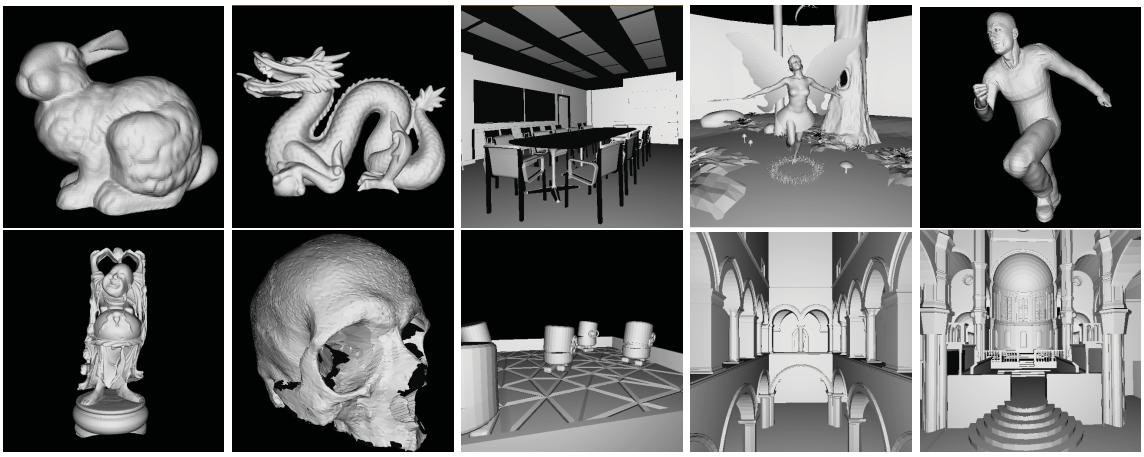


Figure 1: Scenes renderized by our real-time roped-BVH ray caster. From left to right and top to bottom, they are *Bunny* (31.28), *Dragon* (21.60), *Conference Room* (34.52), *Fairy Forest* (22.55), *Runner* (41.67), *Happy Buddha* (24.33), *Skull* (29.64), *Toys* (50.72), *Sponza Atrium* (26.49) and *Sibenik Cathedral* (23.74). The numbers in brackets are the frames per second obtained for a resolution of 1024×1024 on a GeForce 280 GTX.

Abstract

In this paper, we present a real-time ray caster implemented on GPU using CUDA. It uses a BVH augmented with ropes that is traversed with ray packets to speed up performance. We present two algorithms making use of ray packets, `packet-warp` and `packet-block`, which set the packet size to a warp and a block, respectively. We also analyze the influence of the packet size and packet shape by testing several configurations. Finally, we compare the time results we have obtained to previous related papers over a batch of usual scenes.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing; I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processor

Keywords: Real-Time Ray Tracing, Bounding Volume Hierarchies, Graphics Processing Units

*Research supported by the Project CCG08-UCM/TIC-4252.
†e-mail: r.torres@fdi.ucm.es
‡e-mail: pjmartin@sip.ucm.es
§e-mail: agav@sip.ucm.es

1 Introduction

Ray tracing methods generate 2D images from 3D models by shooting rays through the scene. The basic idea is composed of three steps: in the first one, (primary) rays are generated per pixel according to the current camera position; in the second, each ray is tested against the objects of the scene for intersections; and in the last one, the final color of each pixel is calculated, which usually involves the generation of new (secondary) rays to compute shadows, reflections and other effects.

Concerning the second step, which is the one this paper devotes to, many spatial data structures have been proposed to organize the scene in order to speed up spatial search inside. Well-known examples are *uniform grids*, *kd-trees*, *octrees*, *BSP-trees* (*Binary Space Partitioning Trees*) and *BVHs* (*Bounding Volume Hierarchies*). The aim of using them is to rule out ray-object intersection tests as soon as possible. Nevertheless, the price to pay is to integrate a new step into the whole system: the traversal of each ray through the structure.

Recently, we have witnessed the rapid evolution of GPUs, mainly in two aspects. Regarding the underlying architecture, GPUs have specialized for compute-intensive, highly-parallel computation, and modern GPUs widely outperform current CPUs concerning FLOPS [NVIDIA]. With regard to programming, flexibility and commodity have been the main aims in their evolution. On the one hand, many of the stages of the graphics pipeline are now completely programmable. On the other hand, different general-purpose APIs have been developed to hide the graphics details of the pipeline and to make programming on GPU an easier task. CUDA (*Compute Unified Device Architecture*) by [NVIDIA]

is maybe the most widespread representative of these APIs. Briefly, it extends the C programming language with a few expressions to execute code on the graphics hardware.

Despite of the flexibility modern GPUs provide, migrating an algorithm from CPU to GPU is not immediate. Current graphics architectures are very demanding, thus implementing some algorithms on GPU needs a deeper modification of their design. Indeed, the GPU code is sometimes slower than its CPU counterpart. Regarding ray tracing, the main drawback is that recursion is not explicitly available on GPU. Therefore, traversing the usual spatial structures based on trees must be done iteratively. In any case, the high degree of parallelism ray tracing algorithms exhibit and the compute intensity they demand make GPUs suitable to implement them.

In this paper, we have implemented a CUDA-based ray casting with ray packets using a roped BVH. As we will show, we have obtained execution times that are comparable to those presented in recent proposals which simulate recursion by using stacks to traverse the spatial structure. As a second contribution, we have studied the implications of implementing coalesced memory accesses and analyzed the influence of the packet size and the packet shape on performance, giving insights on the implementation details, which are sometimes neglected along the literature.

2 Previous Work

We distinguish three stages in the evolution of GPU-based ray tracing implementations. In the first one, [Carr et al. 2002] exclusively used the GPU compute capability to speed up the ray-triangle intersection step. Later on, [Purcell 2004] integrated all the steps of the system onto the GPU. It used a uniform grid that was traversed by the Amanatides' algorithm [Amanatides and Woo 1987]. A common feature of the traversal of this non-hierarchical structure is that the next node can be guessed without recursion. So, the traversal can be easily implemented on GPU. Nevertheless, for static scenes, these structures are often considered less interesting since they usually perform worse than other hierarchical spatial structures, mainly because they are not adaptive. [Purcell 2004] applied well-known techniques, such as the multi-pass rendering or the early z-culling, in order to avoid the lack of flexibility of old GPUs. Immediately, [Christen 2004] and [Karlsson and Ljungstedt 2004] adapted similar ideas to spatial structures of equivalent complexity.

In the second stage, more efficient and sophisticated structures were studied, such as kd-trees and BVHs. Since kd-trees are the best on CPU [Havran 2000], they were the first to be implemented on GPU. [Foley and Sugerman 2005] presented two traversal algorithms: *kd-tree restart* and *kd-tree backtrack*. In the first one, rays traverse the tree from the root until a leaf is reached. If there is no intersection against the triangles of that leaf, the ray is truncated and the traversal starts from the root again. The aim of this truncation is to ensure that the algorithm will not explore the leaves already traversed. In kd-tree backtrack, the ray is also truncated, but the algorithm goes up in the tree till it reaches an ancestor of the following leaf to be visited in the preorder traversal. Subsequently, different ropes were also included to iteratively implement a preorder tree traversal: [Popov et al. 2007] for kd-trees and [Thrane et al. 2005] for BVHs. The main drawback is that ropes, which are computed off-line, must be stored within the structure increasing its memory footprint.

In the third stage, which includes the most recent approaches, recursion is simulated by stacks: [Horn et al. 2007] for kd-trees and [Günther et al. 2007] for BVHs. The first one is based on the kd-tree restart algorithm, while the latter uses the ray direction to lead the BVH traversal. Although stacks are useful to drive the traversal depending on the ray, each ray must basically keep its own stack [Zhou et al. 2008]. Therefore, the more rays run in parallel, the more memory is demanded. Moreover, in order to handle these stacks quickly with CUDA, they must be stored on shared memory, which is rather small. Hence, there must exist a suitable upper bound for the size of any of the stacks. As a consequence the system strongly depends on the hierarchy and the device.

Concurrently to these stages, another technique has been also applied to speed up performance: the usage of ray packets [Wald et al. 2001]. It is based on the *coherence* property, which usually holds for neighbouring primary rays: they travel across the same nodes inside the structure and intersect the same objects. Therefore, coherent rays are packed mainly to minimize the number of readings from memory, since these data are common to the whole packet. This fact has been used on CPU to take advantage of the memory bandwidth and to highly exploit the memory cache units. On the other hand, the operations that coherent rays require are also the same, thus the overall performance can be also accelerated by using SIMD operations. Although spatial structures and ray packets have been already used to implement real-time ray tracers on CPUs [Wald 2004], leading trends apply GPUs to speed up performance even more. Focusing only on BVHs, [Thrane et al. 2005] does not use packets, while [Günther et al. 2007] does. Hence, roped BVHs traversed with ray packets had not been studied yet.

Rather than focusing on implementing tree traversal on GPU, other authors have studied how to improve the structure synthesis. Thus, many papers have devoted to design faster synthesis methods, which are usually based on the surface area heuristic (SAH) [Goldsmith and Salmon 1987]. Among them, a recent trend proposes to generate the hierarchy directly on the GPU. For example, [Zhou et al. 2008] explains how to build kd-trees with CUDA, which are subsequently traversed by a stack.

Finally, apart from the previous structures that are used to organize the scene, other papers focus on implementing ray-space hierarchies [Amanatides 1984] on the GPU. For example, [Roger et al. 2007] presents a hierarchy for secondary rays, which is traversed by a shader that results in a better GPU occupancy since it avoids divergences. Nevertheless, an extra shader is required to reduce the traversal output, because it can include empty elements. Furthermore, a traversal pass can result in memory overflow if nearby rays are too incoherent.

3 Bounding Volume Hierarchies

A bounding volume (BV) is a closed tridimensional surface that encloses a number of objects of the scene. The ray-BV intersection test must be very fast and Axes-Aligned Bounding Boxes (AABBs) are, actually, the most often choice. The advantage of using BVs is that a ray will not hit any object

contained within a BV, in case it fails to intersect this BV itself.

A bounding volume hierarchy (BVH) [Rubin and Whitted 1980] is a hierarchical scene partitioning structure obtained by grouping BVs into new ones. A BVH is a tree that stores a BV at each node, enclosing the BVs of all of its descendants. Hence, each inner node of the tree has a number of children which can be either inner nodes with associated BVs, or leaf nodes containing objects.

According to [Wald 2004], a uniform representation of the scene makes it easier to develop efficient ray tracers based on BVHs. So, we have made three decisions. First, the only graphics primitives we use are triangles. Second, we only use a single kind of BV: AABBs. And third, we do not use objects since we only consider static scenes. Hence, triangles are handled regardless of the object they belong to.

3.1 BVH construction

The efficiency of a traversal algorithm for ray tracing highly depends on the acceleration structure organizing the scene. For the specific case of BVHs, we follow the construction algorithm presented in [Wald et al. 2007]. It is based on SAH which says that the conditional probability of a ray R to hit a volume S associated to a node, known that it hits the volume associated to its parent, can be estimated by the ratio of the surface areas of the respective BVs:

$$P(hit_R(S)|hit_R(parent(S))) \approx \frac{Area(S)}{Area(parent(S))}$$

This heuristic has been also applied to kd-trees, but its construction is based on the spatial subdivision of the scene instead of on the partition of the objects as BVHs do. The region of the space associated to a given inner node of a kd-tree is divided by a splitting plane into two regions, each one corresponding to its left and right children. Leaf nodes are associated to regions of the space containing a number of triangles. [MacDonald and Booth 1990] applied the SAH to propose an estimation of the cost of a kd-tree as follows:

$$\begin{aligned} Cost_{LF} &= C_{tri} \cdot N_{LF} \\ Cost_{IN} &= C_{BV} + \frac{Cost_L \cdot Area(A_L)}{Area(A)} + \frac{Cost_R \cdot Area(A_R)}{Area(A)} \end{aligned}$$

where C_{tri} is the cost of the ray-triangle intersection, N_{LF} is the number of triangles contained in the leaf LF, A is the volume associated to the inner node IN, C_{BV} is the cost of the ray-volume intersection, $Area(A_L)$ and $Area(A_R)$ are the areas of the volumes A_L and A_R associated to the left and right children of the inner node IN, and $Cost_L$ and $Cost_R$ are the costs of traversing the left and right children, respectively.

[MacDonald and Booth 1990] also proposed a greedy top-down algorithm to build a kd-tree. The idea is to find the splitting plane that produces the best two-leaf tree possible. The splitting candidates are only taken from the planes that contain the faces of the AABBs enclosing each triangle. Concretely, the cost of the tree that is branched by a plane is:

$$C_{BV} + \frac{C_{tri} \cdot N_L \cdot Area(A_L)}{Area(A)} + \frac{C_{tri} \cdot N_R \cdot Area(A_R)}{Area(A)}$$

[Wald et al. 2007] applies the same heuristic to BVHs. The main difference is that splitting candidates are chosen among those aligned planes that cross the centroids of the AABBs enclosing each triangle. As well, the area of each leaf child corresponds to the tightest AABB covering its triangles, and the value C_{BV} is changed into $2 \cdot C_{BV}$ because the two children of binary BVH are always tested for intersection.

3.2 Single ray traversal

The traversal of a ray throughout a BVH corresponds to a preorder traversal of a tree with pruning. When a ray reaches an inner node, we test the ray against the volume associated to the node for intersection. Along the traversal, we keep the minimum distance (t_{min}) for all the triangles already intersected. If the ray hits the BV of an inner node at a distance smaller than t_{min}, we traverse all its children. If the ray miss the node or hits the BV of the node further than t_{min}, we skip its children. For leaves, the ray-triangle intersection test must be evaluated for every triangle of the leaf.

The traversal of a BVH is a recursive process. However, recursive functions cannot be used in CUDA, thus, we extend the BVH by introducing ropes that iteratively guide the recursion process. Therefore, each node is extended with a pointer (rope) that points to the next node in the preorder tree traversal (see Figure 2). The process finishes when the ray finds a rope pointing to *null*.

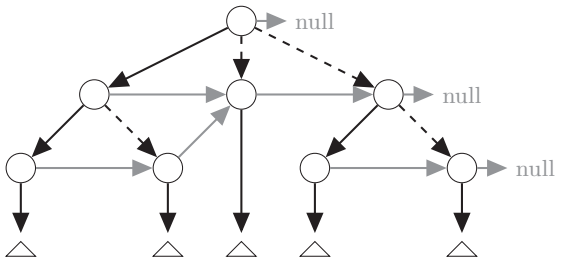


Figure 2: Example of a BVH. Black lines represent references to the children of a node. Dotted lines can be skipped in our rope-based representation, as we will see in Section 4. Grey references are actual ropes. A rope pointing to `null` represents the end of the traversal algorithm.

3.3 Ray packet traversal

The use of ray packets has a significant influence on the performance of real-time ray tracers. As we will see in Section 4, their usage with CUDA improves three aspects. First of all, ray packets are needed to do coalesced readings of triangles and BVH nodes. Second, we need fewer registers since the information that is common to the packet is stored in shared memory. Third, the operations are also common to the whole packet, so the number of divergences will be the minimum, if its rays behave coherently. Unfortunately, processing packets requires additional code that makes the algorithm be more complex.

The traversal algorithm for ray packets is presented in Algorithm 1. N_P and N_R are references to BVH nodes, that respectively denote the next node for the packet and the

```
shared int left[];
t_min = ∞;
N_R = root;
N_P = root;
while (N_P != null) do
    bool active = (N_P == N_R);
    rays in the packet collaborate to bring N_P;
    if (N_P is a leaf) then
        foreach (triangle t in the leaf N_P) do
            rays in the packet collaborate to bring t;
            if (active) then
                test intersection against t;
                update t_min and hitPoint_min;
                N_R = rope(N_P);
    else
        if (active) then
            test intersection against N_P;
            if (there is intersection) then
                N_R = child_left(N_P);
            else
                N_R = rope(N_P);
    if (N_R == child_left(N_P)) then
        left[rayID] = 1;
    else
        left[rayID] = 0;
    ParallelSum(left[0..packetSize-1]);
    if (left[0] > 0) then
        // for some ray, N_R == child_left(N_P)
        N_P = child_left(N_P);
    else
        // for every ray, N_R == rope(N_P)
        N_P = rope(N_P);
```

Algorithm 1: Traversal algorithm of a roped BVH using ray packets.

next node for each ray. Note that N_P is common to all the rays of the packet, while N_R is specific to every single ray. Initially, N_P as well as N_R point to the root of the BVH. We call *active* those rays for which $N_R = N_P$ holds. Every active ray will update its reference N_R, whereas *non-active* rays will wait in their nodes N_R. So, the much time the rays of a packet remain active, the less rays will be waiting and the more efficient the traversal algorithm will be. Nevertheless notice that even non-active rays cooperate to read N_P or the triangle t from global memory. The way rays collaborate is the usual: consecutive threads read consecutive addresses of the data chunk that is being read.

Algorithm 1 works as follows. If N_P is a leaf, an intersection test against its triangles is checked and N_R is updated to the rope. If N_P is inner, active rays check intersection against it and update their references N_R accordingly. Once every reference N_R has been updated, the packet reference N_P is also updated according to the following rules. If all the rays take either the rope or the left child of N_P then N_P is updated to the rope or to the left child, respectively. If there is a divergence inside the packet –that is, some of the rays take the left child while some others take the rope– we update N_P to its left child, since every ray taking the rope will be eventually reached by the others. Therefore, in order to update N_P it is enough to determine whether any ray takes the left child of N_P. This computation is carried out by the parallel reduction `ParallelSum` over the shared array `left`, which is properly initialized for every ray to indicate whether it takes the left child or not.

Finally, the per-ray value t_{min} denotes the distance to the nearest intersection, and $hitPoint_{min}$, the point corresponding to t_{min}. The current t_{min} is used to prune those intersected nodes N_R whose entry distance is beyond it.

4 Implementation Details

Along this paper, we have used two computers: an Intel Pentium 3.0 GHz with an NVidia GeForce8800 GTS (compute capability 1.0) and an Intel Core 2 Quad Q6600 2.4 GHz with an NVidia GeForce 280 GTX (compute capability 1.3).

Following the stream programming model for ray tracing [Purcell et al. 2002], our ray caster has been implemented in three kernels. The kernel Ray Generator (RG) deals with the generation of primary rays from the camera –a standard pinhole camera. The kernel Traversal-Intersection (TI) finds the nearest intersection point for each ray. To do this, it makes ray packets traverse through the BVH and it tests them against triangles for intersection. The kernel Shader (SH) gives the rays color according to the nearest intersection point and light information. The kernel TI takes the longest runtime (about 96% of the total time) because it contains the loop devoted to traverse the BVH.

Unlike [Popov et al. 2007] and [Günther et al. 2007], which use one kernel only, our division in three kernels takes better advantage of the resources in CUDA. The kernels RG and SH respectively spend 12 and 16 registers, therefore the multiprocessor occupancy reaches 100% on both GPUs, whereas TI spends 32 registers, which reaches a lower occupancy (33% on GeForce 8800 GTS and 50% on GeForce 280 GTX). We have proved that the benefit from the execution of three kernels, two out of them with maximum occupancy, is bigger than the overhead time lost in launching those kernels instead of one (we found that our one-kernel algorithm is about 62% slower than our three-kernel algorithm).

Concerning intersection tests, we have implemented the ray-AABB test presented in [Shirley and Morley 2003] and the ray-triangle test presented in [Möller and Trumbore 1997].

4.1 Ray packets

In order to take advantage of the parallelism of rays, a CUDA thread is launched for each ray. Each thread receives a single identification and several consecutive threads make up a ray packet (all ray packets have the same size). To calculate the next N_P, there must exist a communication among threads. CUDA only allows threads to communicate each other if they are in the same CUDA block, hence threads of the same packets have to belong to the same CUDA block. In consequence, there are two obvious ways to implement ray packets: either a packet is made up of the threads in a warp or a packet is made up of the threads in a block. We have called each algorithm `packet-warp` and `packet-block`, respectively.

In `packet-block` the packet size is fixed to the block size. A block is a set of threads that are executed in the same multiprocessor and are able to communicate each other. In this algorithm, the packet size is flexible as long as restrictions of CUDA concerning launching are fulfilled (concretely, regarding the number of available registers per multiprocessor): as the kernel TI requires 32 registers, the blocks can contain

up to 256 threads on the GeForce 8800 GTS and up to 512 on the GeForce 280 GTX, so the packet size is bounded by these two values.

In order to calculate the next N_P, a reduction (`ParallelSum`) is accomplished in shared memory. That reduction is similar to the function `reduction2` of the CUDA SDK [NVIDIA]. In order to prevent race condition, there must be several synchronization points throughout the algorithm.

In `packet-warp` the packet size is fixed to the warp size. A warp is a set of consecutive threads that are all executed at the same time. On both GPUs, the warp size is 32 threads. This approach has several advantages. Firstly, the block size is more flexible since it is not connected to the packet size. Therefore, different configurations can be issued to capitalize on each GPU. Secondly, threads in the same warp are run at the same time, so synchronization is not needed. Finally, that characteristic makes it possible to implement `ParallelSum` more efficiently (see Algorithm 2). Essentially, it is an unrolled version of the algorithm `reduction0` of the CUDA SDK [NVIDIA]. Observe that the `if`-condition and the `__syncthreads` instruction are not needed since every thread in the warp runs at the same time, being faster than `reduction0`. It works for two reasons: first, there are always 16 elements allocated in shared memory next to `left`, and second, their values do not affect the final value (`left[0]`).

```
foreach (thread thID in the packet) do
    left[thID] += left[thID + 1];
    left[thID] += left[thID + 2];
    left[thID] += left[thID + 4];
    left[thID] += left[thID + 8];
    left[thID] += left[thID + 16];
```

Algorithm 2: `ParallelSum` for the `packet-warp` approach.

4.2 Coalesced memory accesses

We think that accesses from/to global memory are the slowest issue of CUDA. So, it is essential to take advantage of the characteristics that CUDA provides to speed up readings to and writings from global memory. CUDA provides coalesced readings and writings, and also readings through textures. In this paper, we have used coalesced accesses.

For compute capability 1.0 cards (GeForce 8800 GTS), two conditions must be fulfilled so that readings and writings are coalesced (cfr. [NVIDIA] for more details). First, data to be accessed must be in memory blocks aligned to 64, 128 or 256 bytes. Second, each block is composed of 16 chunks of n bytes, where n is either 4 for a 64-byte aligned block, 8 for 128-byte or 16 for 256-byte. In addition, the *i-th* thread of a half warp must access the *i-th* chunk. This determines the form of the arrays in memory as we will see next. We have to point out that more modern architectures (compute capability 1.3) have a more flexible access pattern.

Arrays in global memory can be classified into two categories: those that implement data streams and those that hold information of the scene. Concerning the first ones, they are used to pass information between kernels. They are the ray array and the intersection array. Since each component of them spends more than 16 bytes, coalesced accesses require splitting the arrays. Concretely, we have used four:

```
struct __align__(16) t_ray1 {
    float3 origin;     // origin of the ray
    float  padding; }; // padding
struct __align__(16) t_ray2 {
    float3 dir;        // direction
    float  padding; }; // padding

struct __align__(16) t_inter1 {
    float3 point;      // hit point
    float  isInter; }; // boolean flag
struct __align__(16) t_inter2 {
    float3 normal;     // normal
    float  padding; }; // padding
```

Regarding the arrays holding the scene information, we have used two: the array of triangles and the array of BVH nodes. Vertices are not shared among triangles, thus each element in the triangle array stores its three vertices and its normal, which corresponds to 48 bytes. In order to get coalesced accesses we have to align each component to 64 bytes so a padding of 16 bytes is required:

```
struct t_triangle {
    float3  vertex0;
    float3  vertex1;
    float3  vertex2;
    float3  normal;
    float   padding[4]; };
```

Finally, the array of BVH nodes follows the advice of [Smits 1998]: the left child of a node can be stored in the next array position to avoid the use of these pointers (Figure 2). Each node spends 32 bytes, so two nodes are stored in a 64-byte memory block. In a coalesced reading, each packet brings two consecutive nodes into shared memory. Hence, it is possible for the next node to already reside in shared memory when it is required by the algorithm.

```
struct t_bvh_node {
    float   minAABB_X;
    float   minAABB_Y;
    float   minAABB_Z;
    float   maxAABB_X;
    float   maxAABB_Y;
    float   maxAABB_Z__num_triangles;
    int     rope;
    int     index_tri; };
```

The integer `index_tri` codes the following information: if it is greater than 0, the node is a leaf node and the integer corresponds to the index where its list of triangles begins; otherwise, the node is an inner node. The integer `rope` points to the related rope. The value `maxAABB_Z__num_triangles` is the z-coordinate of the AABB's maximum point for inner nodes, and the number of triangles for leaves.

5 Results

A wide range of benchmark scenes have been tested (Figure 1). Each one of them is a representative of a different type of scene. The scenes *Bunny*, *Dragon*, *Happy Buddha*, *Skull* and *Runner* represent scenes with a single mesh including a high number of polygons. *Toys* is a scene with a few numbers of triangles that could represent a snapshot of a simple video game. *Conference Room*, *Fairy Forest*, *Sponza Atrium* and *Sibenik Cathedral* are scenes that could be part of a state-of-the-art video game. It is out of the scope of this work

Scenes	Triangles	Resolutions	GeForce 280 GTX				GeForce 8800 GTS			
			`packet-warp`		`packet-block`		`packet-warp`		`packet-block`	
			FPS	packet shape	FPS	packet shape	FPS	packet shape	FPS	packet shape
Bunny	69,451	512×512	81.11	4×8	57.06	8×8	34.96	4×8	17.61	4×16
		1024×1024	37.28	4×8	20.32	8×8	13.43	4×8	7.06	2×32
Dragon	871,414	512×512	40.66	8×4	21.37	8×8	13.69	8×4	6.42	4×16
		1024×1024	21.60	8×4	10.35	8×8	6.73	4×8	3.17	4×16
Runner	78,029	512×512	86.38	2×16	33.04	1×64	38.60	4×8	19.21	2×32
		1024×1024	41.67	2×16	19.80	1×64	16.91	4×8	9.34	2×32
Fairy Forest	174,117	512×512	49.90	4×8	26.37	8×8	19.31	4×8	10.31	2×32
		1024×1024	22.55	4×8	11.82	2×32	7.97	4×8	4.25	2×32
Happy Buddha	1,087,716	512×512	48.50	2×16	18.70	1×64	16.76	2×16	7.50	1×64
		1024×1024	24.33	2×16	9.51	1×64	8.65	4×8	3.92	2×32
Conference Room	190,947	512×512	92.16	8×4	66.10	8×8	36.66	8×4	22.34	4×16
		1024×1024	34.52	4×8	22.15	2×32	11.99	16×2	7.60	4×16
Sponza Atrium	66,454	512×512	74.48	4×8	48.08	8×8	27.07	4×8	15.76	4×16
		1024×1024	26.49	8×4	15.97	8×8	8.55	4×8	4.95	2×32
Toys	11,141	512×512	122.63	8×4	87.70	8×8	59.22	8×4	35.44	4×16
		1024×1024	50.72	4×8	33.23	8×8	19.80	4×8	12.14	4×16
Skull	102,905	512×512	72.99	4×8	39.35	8×8	28.56	4×8	12.82	2×32
		1024×1024	29.64	4×8	14.32	2×32	10.32	4×8	5.15	2×32
Sibenik Cathedral	80,479	512×512	66.10	4×8	46.12	8×8	24.47	8×4	12.81	8×8
		1024×1024	23.74	4×8	14.79	8×8	7.56	4×8	4.51	4×16

Table 1: The best results of our ray caster. We show the best FPS rates and packet shapes of the `packet-warp` and `packet-block` algorithms on a GeForce 280 GTX and on a GeForce 8800 GTS. They have been obtained at resolutions of 512×512 and 1024×1024, averaging 20 frames on a set of scenes.

to research about shading or illumination algorithms. So, we have implemented a simple Phong diffuse shading with a light point located at the origin of the camera.

The `packet-warp` and `packet-block` algorithms have been implemented and tested over those benchmark scenes. The results of Table 1 are the best results that we have obtained in frames per second (FPS) by averaging 20 frames on different launching configurations. The parameters of these configurations correspond to the constants C_{tri} and C_{BV} of the BVH construction algorithm, the packet size and the packet shape. The packet shape can be changed by varying the shape of the CUDA blocks in both `packet-warp` and `packet-block`. The packet size can be changed only in `packet-block` by varying the size of the CUDA block. The scenes have been tested at resolutions of 512×512 and 1024×1024.

The worst case in the traversal of a roped BVH appears when the ropes drive the rays to test the nodes from far to near. A naive BVH construction produces this case for certain positions of the camera. In order to avoid it, the left and right children are randomly swapped during the construction.

The `packet-warp` algorithm is faster than `packet-block` because of its more efficient TI kernel. This is due to three reasons. First, as explained before, instructions of synchronization are not needed. This avoids both the overhead produced by these instructions and the threads' waiting for synchronization. Second, the more efficient reduction algorithm improves the whole algorithm since it is a fundamental piece within TI. Third, the block size can be changed independently of the packet size; so, it can be conveniently tuned in order to exploit CUDA architecture.

As far as the packet shape, it seems that squarer shapes result in better performance since packet rays are more coherent each other. However, we have seen that not so square packets result in the best performance for certain scenes (*Happy Buddha* and *Runner*). We think that this is due to the elongated shape of these scenes.

We have noted that 64 threads per block is the packet size with the best timing for the `packet-block` algorithm. This is due to two reasons. On the one hand, the number of active rays decreases as the number of rays per packet increases. On the other hand, 64 threads per block reach the upper bound of blocks per multiprocessor, so the occupancy falls sharply if this value decreases.

In order to make the implementation of Algorithm 1 easier, there is not intersection with the leaf nodes' AABB. Instead, each leaf is replaced with two new nodes: an inner and a leaf node. The inner one is the father of the leaf node and contains the AABB, while the leaf node contains the triangles. So, the number of nodes increases and the memory footprint of the BVH is bigger than in other implementations [Günther et al. 2007].

The ropes of a roped BVH establish a traversal order for its nodes. As a result, the number of nodes that each ray or packet has to test for intersection is bigger than a BVH traversal algorithm using a stack. Table 2 compares the number of traversed nodes for two kinds of algorithms: stack-based traversals and roped-based traversals. We have carried out this comparison in two situations: using ray packets and using single rays, thus, we have compared four traversal implementations on CPU: (1) roped-BVH with packets and (2) roped-BVH with single rays, which have been already explained above; (3) stacked-BVH with packets, based on [Günther et al. 2007], and (4) stacked-BVH with single rays, similar to [Günther et al. 2007] but the per-ray entry distances are not stored in the stack.

Table 2 indicates the averaged increase of traversed nodes for single rays (third column) and packets (fourth column).

Scene	Resolutions	single rays rope/stack	packet rope/stack
Bunny	512×512	20.62 %	29.50 %
	1024×1024	17.41 %	23.90 %
Dragon	512×512	36.21 %	48.75 %
	1024×1024	36.12 %	46.90 %
Runner	512×512	14.62 %	20.81 %
	1024×1024	14.66 %	20.20 %
Fairy Forest	512×512	18.17 %	22.47 %
	1024×1024	13.73 %	15.84 %
Happy Buddha	512×512	33.08 %	37.65 %
	1024×1024	37.46 %	42.79 %
Conference Room	512×512	9.78 %	7.09 %
	1024×1024	9.78 %	7.28 %
Sponza Atrium	512×512	-0.09 %	2.31 %
	1024×1024	-0.09 %	2.40 %
Toys	512×512	20.24 %	19.49 %
	1024×1024	20.16 %	19.96 %
Skull	512×512	20.02 %	27.15 %
	1024×1024	63.72 %	78.36 %
Sibenik Cathedral	512×512	37.78 %	23.62 %
	1024×1024	16.02 %	4.62 %

Table 2: Increase of traversed nodes on average.

The values have been obtained for the best configuration as they are in Table 1. The number of traversed nodes varies depending on the scene. The maximum value is reached in *Skull* with 78%. However, these numbers usually ranges from 20% to 40% and it does not seem to be connected to either the amount of triangles or the complexity of the scene. Figure 3 graphically shows that there are groups of rays that traverse more nodes due to ropes.

Table 3 shows timings of other real-time ray tracers. It is difficult to compare our ray caster with other implementations for several reasons. Firstly, some of them use kd-trees (columns 1, 3 and 5), instead of BVHs (columns 2 and 4). Secondly, they run on a different hardware, either on GPU (columns 1, 2 and 3) or on CPU (columns 4 and 5). Thirdly, some of the papers consider dynamic scenes (columns 1 and 4). To make this comparison even more difficult, each implementation has its own characteristics as regards shading.

The work in column 2 (Table 3) is very similar to ours. The authors use a BVH structure, but traversed using a stack. In addition, the card GeForce 8800 GTX they use is similar to our GeForce 8800 GTS. Their results are better when only primary rays are used, but it has to be taken into account that their card is slightly more powerful and their shading seems to be simpler.

The work in column 4 uses a CPU-implemented BVH and applies *frustum culling* of ray packets to improve even more the traversal. The implementation of this technique in GPU is not trivial and we have not used it. However, our results are comparable on the GeForce 8800 GTS and much higher on the GeForce 280 GTX.

The works in columns 1 and 3 present a kd-tree implemented on GPU, whereas the work in column 5, a kd-tree on CPU. The performance timings are similar to ours, getting the work in column 1 the best performance. We point out that the time of secondary rays and kd-tree construction is included in its results.

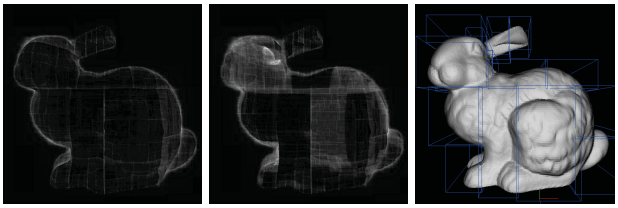


Figure 3: Number of per-ray traversed nodes for the *Bunny* scene. The brightness of a pixel indicates the number of traversed nodes for that pixel's ray. The left image corresponds with the traversal algorithm with stack and single rays. The central image corresponds with the traversal algorithm for the roped BVH with single rays. The right image shows the BVs of a certain BVH level.

6 Conclusions

In this paper, we have presented a real-time ray caster implemented on GPU. It uses a BVH augmented with ropes that is traversed without a stack. It also uses ray packets to speed up the whole algorithm and to improve the performance of the CUDA memory. Ray packets have been implemented through two algorithms: `packet-warp` that states the size of a packet as the size of a warp, and `packet-block` that states it as the size of a block. The performance of the whole algorithm has been tested over a batch of scenes and compared to other works of similar characteristics, obtaining comparable results.

Ray tracing is a highly-parallel algorithm since rays (and ray packets) are independent. Thus, GPUs are hardware suitable to implement ray tracing because of their multi-core architecture. However, implementation details are crucial to get the best performance of the GPU. These details force the application to set certain parameters, such as packet size or memory layout, and complicate the implementation of the algorithm. The performance can fall sharply if those recommendations are not fulfilled.

In spite of the fact that kd-trees are the best structures on CPU, this assertion is not completely true on GPUs. The ray casting presented in this paper has a performance comparable to other works either on CPU or on GPU. In addition, BVHs take a better advantage of big packet sizes than kd-trees. As CUDA requirements force the application to have big packet sizes, BVHs are a good recommendation for GPU ray tracers.

The usage of ropes instead of a stack increases the memory footprint but prevents implementation of the stack in shared memory. Moreover, ropes can drive the ray tracer to the worst case: the case when each ray tests the BVH nodes from far to near. Nevertheless, the number of traversed nodes in excess is low enough for roped BVHs to consider them as an interesting acceleration structure for ray tracing.

References

Amanatides, J., and Woo, A. 1987. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, 3–10.

Amanatides, J. 1984. Ray tracing with cones. *SIGGRAPH Comput. Graph. 18*, 3, 129–135.

Scene	1	2	3	4	5	our ray caster GeForce 280 GTX	our ray caster GeForce 8800 GTS
Bunny	-	-	12.7/5.9	-	-	37.28 (4×8)	13.43 (4×8)
Toys	32.0	-	-	30.5	-	50.72 (4×8)	19.80 (4×8)
Fairy Forest	6.4	13.2/4.8	10.6/4.0	7.7	-	22.55 (4×8)	7.97 (4×8)
Conference Room	-	16.0/6.1	16.7/2.7	9.3	19.5/15.6	34.52 (4×8)	11.99 (16×2)
Runner	-	-	-	21.4	-	41.67 (2×16)	16.91 (4×8)

Table 3: Comparison of the performance of our ray caster with other papers. FPS in column 1 are from [Zhou et al. 2008] on a GeForce 8800 ULTRA. Column 2 are from [Günther et al. 2007] on a GeForce 8800 GTX. Column 3 are from [Popov et al. 2007] on a GeForce 8800 GTX. Column 4 are from [Wald et al. 2007] on a AMD Opteron 2.6 GHz. Column 5 are from [Reshetov et al. 2005] on an Intel Pentium 4 3.2 GHz with multithreading. The last two columns are from our ray caster implemented on a GeForce 280 GTX and on a GeForce 8800 GTS, respectively. All the timings were obtained at a resolution of 1024×1024.

Carr, N. A., Hall, J. D., and Hart, J. C. 2002. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 37–46.

Christen, M. 2004. Implementing ray tracing on gpu. Tech. rep., Diploma thesis, University of Applied Sciences.

Foley, T., and Sugerman, J. 2005. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, 15–22.

Goldsmith, J., and Salmon, J. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl. 7*, 5, 14–20.

Günther, J., Popov, S., Seidel, H.-P., and Slusallek, P. 2007. Realtime ray tracing on gpu with bvh-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, 113–118.

Havran, V. 2000. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. 2007. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 167–174.

Karlsson, F., and Ljungstedt, C. J. 2004. *Ray tracing fully implemented on programmable graphics hardware*. Master's thesis, Chalmers University of Technology.

MacDonald, D. J., and Booth, K. S. 1990. Heuristics for ray tracing using space subdivision. *Vis. Comput. 6*, 3, 153–166.

Möller, T., and Trumbore, B. 1997. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools 2*, 1, 21–28.

Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. 2007. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (Sept.), 415–424. (Proceedings of Eurographics).

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July), 703–712.

Purcell, T. J. 2004. *Ray tracing on a stream processor*. PhD thesis, Stanford, CA, USA. Adviser-Patrick M. Hanrahan.

Reshetov, A., Soupikov, A., and Hurley, J. 2005. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, 1176–1185.

Roger, D., Assarsson, U., and Holzschuch, N. 2007. Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the gpu. In *Rendering Techniques 2007*, 99–110.

Rubin, S. M., and Whitted, T. 1980. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, ACM, 110–116.

Shirley, P., and Morley, R. K. 2003. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA.

Smits, B. 1998. Efficiency issues for ray tracing. *Journal of Graphics Tools 3*, 1–14.

NVIDIA Nvidia website `www.nvidia.org`.

Thrane, N., Simonsen, L. O., and Ørbæk, A. P. 2005. A comparison of acceleration structures for gpu assisted ray tracing. Tech. rep.

Wald, I., Benthin, C., Wagner, M., and Slusallek, P. 2001. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, vol. 20, 153–164.

Wald, I., Boulos, S., and Shirley, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transaction on Graphics 26*, 1, 6.

Wald, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University.

Zhou, K., Hou, Q., Wang, R., and Guo, B. 2008. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5, 1–11.

Ray Tracing en CUDA usando una BVH hilvanada

R. Torres de Alba, P. J. Martín de la Calle, A. Gavilanes Franco

Departamento de Sistemas y Computación, Universidad Complutense de Madrid, Madrid, España
{r.torres@fdi, pjmartin@sip, agav@sip}.ucm.es

Resumen

En este artículo presentamos una implementación en GPU de un ray tracing en tiempo real que usa paquetes de rayos y una BVH como estructura de aceleración. Para ello, hemos aprovechado características de la tecnología CUDA tales como la memoria compartida, las lecturas y escrituras fusionadas y las lecturas a través de la caché de texturas. Además, para poder recorrer la BVH sin recursión, ésta ha sido aumentada con hilvanes que marcan el orden de recorrido de los nodos. También incluimos un análisis de nuestro algoritmo sobre diversas escenas.

1. Introducción

El *ray tracing* consiste en una familia de algoritmos que se encargan de generar imágenes bidimensionales a partir de la representación tridimensional de una escena. Todos ellos tienen en común que realizan esta tarea lanzando rayos por la escena misma.

En todos estos algoritmos se distinguen fundamentalmente tres etapas. En la primera, se generan rayos (conocidos como primarios) a partir de un modelo de cámara. En la segunda, cada rayo busca la primera intersección con los objetos de la escena. En la tercera, se generan nuevos rayos (conocidos como secundarios), que sirven para calcular el color en dicho punto, en caso de intersección.

Para la segunda etapa (la que más concierne a este trabajo) existen diferentes algoritmos que la resuelven. El enfoque más simple es el algoritmo de fuerza bruta, que consiste en calcular la intersección de cada rayo con todos los objetos. Sin embargo, en una situación real, el número de objetos y de rayos es muy elevado, y usarlo enlentecería enormemente el algoritmo completo. De hecho, Whitted [1] observó que esta etapa consumía más del 70 % del tiempo total cuando se empleaba la fuerza bruta.

Para solucionar este problema se han propuesto diferentes estructuras espaciales de organización de la escena. Entre ellas se encuentran las rejillas uniformes, los *KD-trees*, los *Octrees*, los *BSP-trees* (*Binary Space Partitioning Trees* o árboles de Particionamiento Binario del Espacio) y las *BVHs* (*Bounding Volume Hierarchies* o Jerarquías de Volúmenes Delimitadores). Todas estas estructuras se encargan de organizar la escena de tal forma que puedan descartarse muchos objetos de la prueba de intersección con rapidez. El precio a pagar es la adición de una nueva etapa: la etapa de recorrido del rayo a través de la estructura.

Otro instrumento útil en la mejora del rendimiento es el uso de paquetes de rayos. En este caso, se trata de aprovechar la alta coherencia que tienen los rayos con orígenes y direcciones semejantes. Se dice que dos rayos son coherentes si se comportan igual, es decir, si intersecan con los mismos volúmenes de la estructura y con los mismos objetos. Eso facilita que ciertas operaciones puedan ser optimizadas para estos paquetes de rayos. En [2] se usa este hecho para aprovechar las operaciones SIMD de determinadas CPUs y aumentar en un factor de 3.5 los algoritmos desarrollados para paquetes de rayos.

Por otro lado, las GPUs (*Graphics Processing Units*) son dispositivos hardware especializados en implementar la tubería gráfica. En un principio, la mayoría de sus partes eran fijas y la personalización quedaba reducida a unos pocos parámetros. Actualmente se ha dado un gran impulso a su flexibilidad y varias etapas de la tubería han llegado a ser completamente programables. Además, hoy en día, debido a las características de paralelismo y de intensidad de cómputo que requiere los procesos de la tubería gráfica, la potencia de cálculo de las GPUs ha superado en mucho a la de las CPUs [3].

Ambas propiedades (programabilidad e intensidad de cómputo) han potenciado el uso de las GPUs en aplicaciones no relacionadas con el algoritmo al que estaban dedicadas en un principio. Esto ha dado lugar a una rama conocida como *GPGPU* (*General-Purpose Computing on GPU*) [4]. Así, se ha probado la efectividad de las GPUs en aplicaciones relacionadas con el tratamiento de grandes volúmenes de información donde el cómputo por dato es independiente de los demás. Este hecho ha tenido tanto éxito que se han desarrollado diferentes APIs para la programación genérica de GPUs. Uno de ellos, el que aplicamos en este trabajo, es el llamado CUDA [3], desarrollado por NVIDIA, que consiste en un lenguaje parecido a C.

Uno de los grandes inconvenientes de la programación en GPU es la imposibilidad de programar directamente con recursión. Esto es consecuencia de que no existe una pila de llamadas implícita, sino que ésta debe ser implementada y manejada explícitamente por el programador. Por este motivo, no es trivial el paso a la GPU de algoritmos que son tan fácilmente programados en CPU.

Artículo publicado en Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones (ANACAP) 2008.
ISBN 978-84-691-7426-5.

En el caso concreto de los algoritmos de *ray tracing*, la implementación en GPU de las estructuras espaciales mencionadas y de los algoritmos que las recorren resulta complicado. Aún así, el alto grado de paralelismo y la intensidad de cálculo, que son características comunes a estos métodos, hacen atractivo el uso de las GPU para su implementación.

Aunque las anteriores estructuras espaciales ya se han desarrollado para CPU, éste no es el caso para las GPUs debido a su reciente uso como hardware de propósito general. Además, el comportamiento de estas estructuras en GPU es crítico, dado que la implementación en GPU persigue una mejora del rendimiento total del sistema. Este trabajo pretende progresar en el análisis de estas estructuras en GPU, ya que presenta un método de *ray tracing* que usa una estructura espacial BVH hilvanada, implementada en CUDA, que incluye el uso de paquetes de rayos.

2. Trabajos previos

Ya se han publicado varios trabajos para acelerar el *ray tracing* en GPU [5], [6] y [7]. Uno de los primeros, el *Ray Engine* [8], emplea la GPU para acelerar la etapa de intersección rayo-triángulo. Básicamente, se usa para ejecutar la intersección de un lote de rayos con un triángulo. Ya que el código implicado no posee muchos saltos y todos los hilos realizan las mismas operaciones, se trata de una tarea muy adecuada para su implementación en GPU.

[9] introduce todas las partes del ray tracing en la GPU, ahorrándose la sobrecarga originada por el trasiego de información entre CPU y GPU. Para evitar la necesidad de una pila, emplea la estructura de rejilla uniforme recorrida con el algoritmo de Amanatides [10], en el que cada rayo calcula en cada etapa el siguiente volumen que atravesará. Hace uso de técnicas habituales en la programación en GPUs como la multipasada o el *early z culling* para solucionar las carencias de flexibilidad que las GPUs tenían entonces.

Se ha probado que los *KD-trees* son la estructura más eficiente en CPU [11]. Sin embargo, esto no está claro en el caso de las GPUs. Se han presentado varios trabajos que implementan en GPU su recorrido sin pila. En [12] encontramos dos: *KD-tree restart* y *KD-tree backtrack*. En *KD-tree restart*, cuando un rayo alcanza una hoja y no hay intersección, se modifica su tamaño y se comienza a recorrer el *KD-tree* desde la raíz otra vez. Esta modificación hace que no se llegue de nuevo al mismo nodo hoja, sino al siguiente en la dirección del rayo. En *KD-tree backtrack*, al igual que en *KD-tree restart*, el tamaño del rayo se modifica al llegar a una hoja, pero esta vez no se vuelve a recorrer el árbol desde la raíz, sino desde el nodo interno más cercano que interseca con el rayo, volviendo a descender a continuación.

En [13], el *KD-tree* es aumentado con hilvanes para que se pueda recorrer más eficientemente sin pila. Además, también se presenta un algoritmo de recorrido que usa paquetes de rayos.

Por último, se han presentado trabajos para recorrer las BVH. En [14] se muestra una implementación en GPU de una BVH hilvanada, pero que no se recorre con paquetes de rayos. En [15] se recorre una BVH con paquetes y con una pila implementada explícitamente en memoria compartida de CUDA.

3. Jerarquía de Volúmenes Delimitadores (BVH)

Una BVH es un árbol de volúmenes delimitadores que ordena jerárquicamente los objetos de la escena. Un volumen (delimitador) es cualquier espacio acotado por de una superficie tridimensional cerrada. El objetivo de usar una BVH es elegir volúmenes cuya intersección con el rayo sea fácil de calcular, tales como esferas o cubos alineados con los ejes (AABBs, de *axis-aligned bounding boxes*). Cada nodo del árbol, ya sea una hoja o un nodo interno, tiene asociado un volumen. El volumen de cada hoja recubre completamente un objeto de la escena. A su vez, los volúmenes de los nodos internos recubren completamente los volúmenes de todos sus hijos. La ventaja de usar esta estructura es que, si se cumple que un rayo no interseca con un volumen, tampoco lo hará con los objetos contenidos en él, por lo que la correspondiente rama puede ser descartada de la prueba de intersección.

Tal y como se recomienda en [16], una representación uniforme de la escena facilita el desarrollo de implementaciones eficientes. Por ello hemos tomado tres decisiones. La primera es representar los objetos como mallas de triángulos, ya que aproximan fielmente cualquier tipo de superficie. La segunda es emplear un único tipo de volumen delimitador. En este trabajo se ha elegido la esfera porque la prueba de intersección es muy rápida. La tercera es no considerar la noción de objeto, ya que, por tratarse de escenas estáticas, resulta más interesante manejar los triángulos de las mallas independientemente. En efecto, si en la escena se diera el caso de que hubiera un objeto de alta poligonalización, la prueba de intersección con un rayo supondría probar con todos sus polígonos, lo que representaría un gasto excesivo, no siendo adecuado para entornos de tiempo real. En consecuencia, cada hoja contendrá un único triángulo.

3.1. Construcción

Para construir una BVH hemos elegido el algoritmo de [17]. Este algoritmo sigue un enfoque de arriba-abajo para construir BVHs con un número de hijos desconocido a priori. Construye en tiempo $O(nlog(n))$, (donde n es el número de objetos de la escena, una BVH usando la siguiente heurística: la probabilidad de intersección de un rayo r con el volumen de un nodo n condicionado por la intersección de ese mismo rayo con la raíz de la BVH se aproxima con el área del volumen del nodo dividido por el área del volumen de la raíz. En términos matemáticos:

$$p(hit_r(n)/hit_r(root)) \approx \frac{area(n)}{area(root)}$$

En este contexto, [17] establece que el coste de inserción de un objeto en un nodo interno es

$$(area_{new} - area_{old})k + area_{new}$$

y para un nodo hoja es

$$2area_{new}$$

donde k es el número de hijos del nodo, $area_{old}$ es el área del volumen asociado al nodo antes de la inserción, y $area_{new}$ es el área después de la inserción.

El algoritmo es como sigue. Los objetos se ordenan aleatoriamente y se elige, entre ellos, a uno como raíz. Posteriormente se van insertando el resto de los objetos, uno tras otro, de la siguiente forma. Para cada objeto se busca la ruta desde la raíz a una hoja eligiendo, en cada nivel, el hijo cuyo coste de inserción es menor. Una vez obtenida dicha ruta, se elige como padre el nodo de la ruta que minimiza el coste de inserción.

El algoritmo adolece de ciertas ambigüedades. Primero, el orden de introducción de los elementos determina el árbol resultante. [17] afirma que la BVH que se construye es mejor si el orden de inserción de los objetos es completamente aleatorio. Segundo, el coste de intersección de un objeto puede ser el mismo para varios hijos de un mismo nodo. Esto sucede en los primeros niveles, cuando los volúmenes llegan a ser tan grandes como la escena completa. La elección del nodo a seguir en estos casos se resuelve de nuevo aleatoriamente.

3.2. Recorrido para un único rayo

El algoritmo de recorrido de un rayo por una BVH consiste en un recorrido del árbol en preorden. En caso de que ese rayo no interseque con un nodo se pasará al siguiente, según este orden, nodo que se encontrará en la cima de la pila. Si se ha llegado a un nodo hoja que tiene un volumen con el que sí interseca, se procederá a probar la intersección con el triángulo contenido. En caso de que haya intersección, se actualizará la intersección más cercana que se lleva hasta ese momento y se continuará recorriendo el árbol. El algoritmo termina cuando el árbol ha sido explorado por completo. No se puede detener la búsqueda en la primera intersección porque no hay garantía de que el orden en que se atraviesan los volúmenes vaya del objeto más cercano a la cámara al más lejano.

Para recorrer una BVH sin pila es necesario añadir cierta información. En este trabajo usamos un hilván para apuntar al siguiente nodo a visitar en preorden, en caso de que el nodo actual deba descartarse (Figura 1).

En el algoritmo de recorrido con hilvanes (Figura 2) no es necesaria una pila porque los hilvanes determinan el orden de recorrido de los nodos. En caso de que un rayo sí interseque con un nodo interno, el siguiente nodo será su hijo izquierdo, igual que en el caso anterior. En caso de que un rayo no interseque con un nodo interno, el siguiente nodo será el nodo al que apunta su hilván. Si el nodo es hoja entonces el siguiente nodo siempre será señalado por su hilván. El algoritmo termina cuando se encuentre un hilván

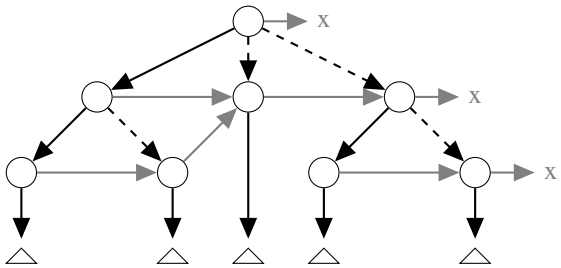


Figura 1. *Ejemplo de una BVH. Las líneas negras representan las referencias a los hijos de un nodo. Además, si la línea está punteada entonces no es necesario implementar esa referencia. Las referencias grises representan los hilvanes. Si un hilván apunta a* `x` *significa que es* `null`*.*

que apunta a *null*, es decir, no existen más nodos por probar y el árbol ha sido recorrido completamente.

```
N_R = raíz;
while (N_R != null) do
    traer nodo N_R;
    probar intersección con el nodo N_R;
    if (hay intersección con N_R) then
        if (N_R es hoja) then
            traer triángulo t;
            probar intersección con triángulo t;
            actualizar t_min e idTri;
            N_R = hilván(N_R);
        else
            N_R = hijo_izquierdo(N_R);
    else
        N_R = hilván(N_R);
```

Figura 2. *Algoritmo de recorrido de una BVH hilvanado que no usa paquetes de rayos. t_{min} representa el tiempo del rayo de la intersección más cercana e $idTri$ es el triángulo con el que se ha producido dicha intersección.*

3.3. Recorrido con paquetes de rayos

El motivo de usar paquetes de rayos es doble. Por un lado, aprovechar la coherencia de los rayos sirve para que la cantidad de información que debe traerse de memoria (nodos y triángulos) sea menor. Por otro, debido a las características de CUDA, como se verá en la Sección 4, los rayos del mismo paquete pueden colaborar al traerse la información de memoria global una única vez para todo el paquete. Desgraciadamente, el tratamiento de los paquetes añade código que complica el algoritmo. Decidir si merece la pena o no el uso de paquetes es un asunto pendiente.

Sean N_P y N_R referencias a nodos de la BVH. N_P determinará el siguiente nodo del paquete de rayos, mientras que N_R señalará al siguiente nodo de cada rayo. Cada paquete posee un único N_P y cada rayo posee un único N_R, es decir, cada rayo tiene su N_R más el N_P del paquete al que pertenece. Al principio, tanto N_P como N_R van a apuntar al nodo raíz de la BVH. Los rayos activos se definen como aquellos rayos para los que se cumple que $N_R = N_P$. La idea es que sólo aquellos rayos que están activos van a actualizar su N_R, mientras que los que no lo están se quedan esperando en su nodo. Sin embargo, todos los rayos del paquete colaboran para traer información, en concreto, traerán el nodo N_P y el triángulo de N_P, si se trata de una hoja.

El algoritmo de recorrido (Figura 3) se compone de dos partes. En la primera, cada rayo activo prueba la intersección con su nodo N_R y lo actualiza. En la segunda, todos los rayos actualizan N_P, pero lo hacen al mismo valor, por lo que N_P será el mismo para todos.

```
N_R = raíz;
N_P = raíz;
while (N_P != null) do
    bool activo = (N_P == N_R);
    colaborar para traer el nodo N_P;
    if (N_P es hoja) then
        colaborar para traer el triángulo t;
        if (activo) then
            probar intersección con t;
            actualizar t_min e idTri;
            N_R = hilvan(N_P);
    else
        if (activo) then
            probar intersección con N_P;
            if (hay intersección con N_P) then
                N_R = hijo_izquierdo(N_P);
            else
                N_R = hilván(N_P);
    if (existe algún rayo con
    N_R==hijo_izquierdo(N_P)) then
        N_P = hijo_izquierdo(N_P);
    else
        N_P = hilván(N_P);
```

Figura 3. *Algoritmo de recorrido de una BVH con paquetes de rayos. Los valores t_{min} e $idTri$ son los mismos que en la Figura 2.*

Aplicar la prueba de intersección rayo-esfera en los nodos hoja supondría ahorrar la prueba de intersección rayo-triángulo (y, por tanto, traer el triángulo) cuando ningún rayo activo del paquete cortara la esfera. Sin embargo, esto implicaría que cada rayo activo debiera conocer el resultado de la intersección de los demás rayos activos de su paquete con ella y, por tanto, se requeriría una comunicación entre hilos implementada mediante escrituras en memoria compartida. Todo ello complicaría demasiado el código y, posiblemente, el rendimiento. Por esto, hemos decidido que, cuando un paquete de rayos alcanza una hoja, probamos directamente con el triángulo que contiene y obviamos la esfera que lo delimita.

Para actualizar N_P pueden darse dos situaciones. En la primera, todos los rayos activos actualizan su N_R al mismo valor. En ese caso, se asigna el valor del nuevo N_R a N_P. En la segunda se produce una divergencia en el paquete, es decir, que unos rayos activos actualizan su N_R como el hijo izquierdo y otros como el hilván. En este último caso existen dos opciones, actualizar N_P al hijo izquierdo o al hilván. Sin embargo, está garantizado que si un rayo viaja por el hijo izquierdo, posteriormente llegará al nodo al que se llegaba por el hilván, ya que la BVH se explora por completo. Por este motivo, en caso de divergencia N_P se actualiza a su hijo izquierdo, ya que los rayos que se han ido por el hilván se convertirán en activos cuando el nodo del paquete los alcance.

4. Implementación

Hemos implementado en CUDA un algoritmo de *ray tracing*, basado en los elementos descritos en la sección anterior, que consta de tres *kernels* (ver Tabla 1). El primero, llamado *GeneraRayos*, se encarga de generar los rayos primarios a partir de la información de la cámara –una cámara *pinhole* estándar. El segundo, llamado *Recorrido-Intersección*, se encarga de hacer viajar los paquetes de rayos por la BVH, y de encontrar la intersección más cercana. El tercero, llamado *Shader*, cambia el valor del rayo por uno que corresponde a una reflexión perfecta en aquellos lugares donde se ha producido una intersección. El número de reflexiones que se lanzan está implementado mediante un bucle en CPU controlado por un parámetro. Concretamente, el bucle ejecuta los *kernels* de *Recorrido-Intersección* y *Shader* tantas veces como indique el parámetro.

Para aprovechar que se realizan las mismas operaciones para todos los rayos y que estos son independientes, se lanzar un hilo por cada rayo. Cada rayo va a tener su origen en el centro de la cámara y va a pasar por el centro de cada píxel. En consecuencia, el número de hilos coincide con el número de píxeles de la imagen final y es posible que se originen efectos de *aliasing*.

Según la terminología CUDA, un *warp* es un conjunto de hilos consecutivos que ejecutan las mismas operaciones en paralelo de forma SIMD, es decir, ningún hilo se adelanta a otro. En las tarjetas que hemos usado, el tamaño de *warp* es de 32 hilos consecutivos. Nosotros hemos establecido el tamaño del paquete en 32 rayos (el mismo que el de *warp*) por varias razones. La principal es la sincronización. Debido a que cada hilo escribe en memoria compartida, es necesaria una sincronización a nivel de paquete. Por el contrario, CUDA sólo ofrece sincronía a nivel de bloque. Asignar el tamaño del paquete al del *warp* garantiza que las operaciones se ejecutan a la vez y, por tanto, son síncronas. Otra razón son las peticiones a memoria. Como veremos a continuación, se pueden fusionar peticiones a memoria si 16 hilos consecutivos leen o escriben de memoria global bajo ciertas condiciones que se mencionan más adelante. Éstas se consiguen fácilmente si tenemos paquetes cuyo tamaño sea múltiplo de 16.

Creemos que la memoria es el cuello de botella en la programación con CUDA. Por eso vamos a aprovechar las características que CUDA ofrece para aumentar su rendimiento mediante lecturas y escrituras fusionadas (*loads and stores coalesced*) y lecturas a través de texturas (*texture fetches*).

La escena completa (triángulos y BVH) se almacena en tres arrays: uno para los vértices de los triángulos, otro para las normales y otro para los nodos de la BVH. La BVH se representa de la misma forma que en [14]: el hijo izquierdo de un nodo ocupa la posición consecutiva a la del padre. Cada elemento del array de vértices corresponde a tres vértices que forman un único triángulo, por ello no existe compartición de vértices entre distintos triángulos. El número de normales es el mismo que el de vértices y existe una correspondencia uno a uno entre vértice y normal.

Las lecturas a través de texturas son usadas en el *kernel*

Kernel	Núm registros	Mem compartida	Conf. rejilla	Conf. bloque	Ocupación
GR	13	0 bytes	40x30=1200 bloques	16x16=256 hilos	67 %
RI	32	1024 bytes	80x15=1200 bloques	8x32=256 hilos	33 %
SH	22	0 bytes	40x30=1200 bloques	16x16=256 hilos	33 %

Tabla 1. *Información detallada sobre diversos aspectos de los kernels para una resolución de 640x480. Los kernels son: GR - GeneraRayos, RI - Recorrido-Intersección y SH - Shader*

Shader para leer los vértices y las normales. La proximidad de los datos que lee justifica el uso de texturas ya que su acceso está cacheado.

Para que se puedan producir lecturas de memoria fusionadas son necesarios dos requisitos. El primero es que hilos consecutivos traigan información consecutiva. El segundo es que la información que traigan está alineada en bloques de 64 bytes. Por ello hacemos que cada elemento del array de nodos y del de vértices ocupe 16 floats (=64 bytes). Como resultado obtenemos las siguientes estructuras:

```
struct nodo {
  float3 centro; /* centro esfera */
  float  radio;  /* radio esfera */
  int    hilvan; /* hilvan */
  int    indice_triangulo; /* si es hoja */
  float  padding[10];
};
struct vertice {
  float3 v0; /* vertice 0 */
  float3 v1; /* vertice 1 */
  float3 v2; /* vertice 2 */
  float  padding[7];
};
```

Desgraciadamente, el cumplimiento de estos requisitos supone añadir mucha información inútil, llegando a que sólo el 56.25 % es útil en el array de vértices y el 37.50 % en el de nodos.

Para el resto de información hacen falta tres arrays más, dos para guardar los rayos y uno para las intersecciones:

```
struct __align__(16) rayo1 {
  float3 origen; /* origen del rayo */
  float  t_max;  /* tiempo maximo */
};
struct __align__(16) rayo2 {
  float3 dir;    /* direccion */
  float  factor; /* factor */
};
struct __align__(8) interseccion {
  float  tiempo; /* tiempo */
  int    idTri;  /* id del triangulo */
};
```

Obsérvese que, nuevamente, el uso de las lecturas y escrituras fusionadas supone descomponer el array de rayos en `rayos1` y `rayos2`. De esa forma el alineamiento se establece a 16 bytes para `rayos1` y `rayos2` y a 8 bytes para `intersecciones`.

El trasiego de información entre la CPU y la GPU es mínimo. Sólo se envían pequeñas cantidades de bytes, por ejemplo, la información de la posición de la cámara. Otras estructuras más grandes, como la información de la escena, se envía una vez al comienzo de la aplicación y reside en memoria global durante todo el proceso.

El algoritmo de intersección rayo-triángulo se basa en [18] aunque está adaptado a GPU para evitar la gran cantidad de saltos que contiene el código.

La memoria compartida sólo se usa en el *kernel Recorrido-Intersección*. La cantidad de memoria usada (4 bytes por hilo) sirve para guardar la información de los nodos y vértices, así como para para calcular el siguiente nodo N_P del paquete. Como se hace en momentos diferentes, se puede usar la misma memoria para almacenar toda esa información anterior. De la manera anteriormente descrita, los rayos colaboran para guardar los nodos y los vértices en la memoria compartida, que posteriormente se leerá para cargarse en registros. No se producen conflictos en los bancos de memoria compartida. Por un lado, cuando se accede a memoria global de forma fusionada, cada uno de los 16 hilos del *half warp* escribe en su propio banco, por lo que no se generan conflictos. Por otro lado, cuando la información ya se encuentra en memoria compartida, todos los hilos leen sucesivamente del mismo banco para traerse esa información a registros, por lo que tampoco hay conflictos.

Como ya se ha mencionado, cada rayo mantiene en registros el valor del siguiente nodo del rayo N_R y del paquete N_P. El nodo del paquete se actualiza escribiendo en memoria compartida. Para ello, cada rayo activo escribe en un array residente memoria compartida un 1 si su siguiente nodo N_R es el hijo izquierdo del nodo del paquete, y un 0 si es el de su hilván. Posteriormente se calcula la suma de todos estos valores mediante una reducción [19]. Si el valor es mayor que 0 entonces se cumple que existe algún rayo que ha tomado como dirección el hijo izquierdo. En ese caso todos actualizan el valor del nodo del paquete N_P al hijo izquierdo. En caso de que sea 0, lo actualizan al del hilván.

5. Resultados

Hemos realizado un estudio de nuestro algoritmo de recorrido de BVHs con paquetes de rayos sobre un par de escenas. La escena de los cubos (Figura 4) consiste en una habitación con un número de cubos fijado por el usuario y colocados aleatoriamente. Tanto los cubos como la habitación están compuestos por 12 triángulos cada uno. La escena del conejo (Figura 5) consiste en el conejo de Stanford que pertenece al repositorio [20].

Las escenas han sido probadas sobre un ordenador con CPU Intel Pentium 4 a 3 GHz, 1 GB de memoria RAM y una tarjeta GeForce 8800 GTS con 640 MB y CUDA 1.1. El sistema operativo ha sido Ubuntu 7.10 con kernel 2.6.22-14-generic.

El rendimiento medio en FPS (*frames per second*) de la ejecución de estas escenas se muestran en la Tabla 2. Para analizar el rendimiento de nuestra BVH respecto al número de triángulos, hemos realizado mediciones en la escena de los cubos (Figura 6). Podemos observar que la curva se comporta linealmente cuando el número de cubos crece ex-

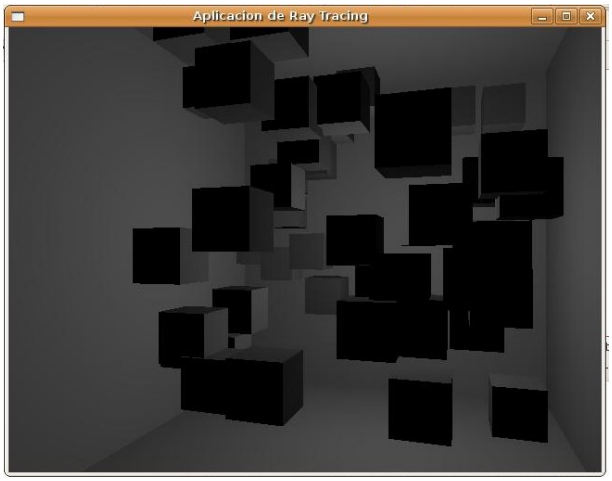


Figura 4. *Captura de la escena de los cubos.*

Figura 5. *Captura de la escena del conejo de Stanford.*

ponencialmente desde 16 has 512, lo que corresponde con la estructura arbórea de las BVHs.

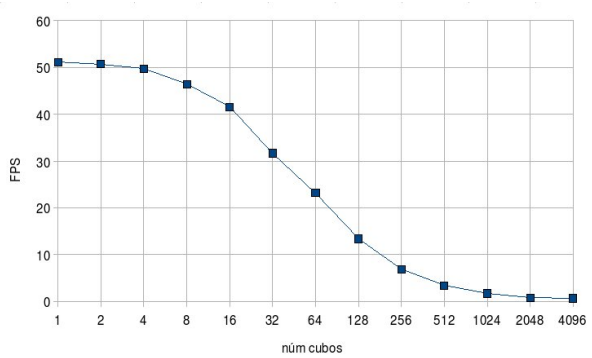


Figura 6. *Representacion gráfica del rendimiento de las escenas de los cubos. En el eje de ordenadas encontramos los FPS resultantes. El eje de abcisas representa el número de cubos de la escena representados en escala logarítmica.*

6. Trabajo Futuro

Todavía quedan muchas cosas por hacer. Algunas de ellas suponen mejoras que aumentarán con seguridad el rendimiento del algoritmo. Por ejemplo, es posible descartar un nodo (y, por consiguiente, todos sus hijos) cuando el punto de intersección de su volumen con el rayo sea mayor que el punto de intersección actual.

Escena	Número triángulos	FPS1	FPS2
Cubos_0	$2^0 + 1$ cubos = 24 tri	51.1	49.3
Cubos_1	$2^1 + 1$ cubos = 36 tri	50.7	50.3
Cubos_2	$2^2 + 1$ cubos = 60 tri	49.7	49.2
Cubos_3	$2^3 + 1$ cubos = 108 tri	46.4	40.9
Cubos_4	$2^4 + 1$ cubos = 204 tri	41.5	36.7
Cubos_5	$2^5 + 1$ cubos = 396 tri	31.7	27.1
Cubos_6	$2^6 + 1$ cubos = 780 tri	23.2	19.0
Cubos_7	$2^7 + 1$ cubos = 1548 tri	13.4	12.0
Cubos_8	$2^8 + 1$ cubos = 3084 tri	6.9	6.5
Cubos_9	$2^9 + 1$ cubos = 6756 tri	3.4	3.3
Cubos_{10}	$2^{10} + 1$ cubos = 12300 tri	1.7	1.6
Cubos_{11}	$2^{11} + 1$ cubos = 24588 tri	0.8	0.8
Cubos_{12}	$2^{12} + 1$ cubos = 49164 tri	0.7	0.4
Conejo de Stanford	69451 tri	0.5	0.4

Tabla 2. *Medidas obtenidas con nuestro algoritmo de ray tracing. Para las escenas de los cubos, cada cubo está formado por 12 triángulos. El número de cubos es potencia de dos más el cubo que representa la habitación. La columna FPS1 corresponde a los frames por segundo medios de cada escena, usando padding en el array de vértices para que cada triángulo esté alineado a 64 bytes. La columna FPS2 es igual que FPS1 pero sin dicho padding.*

Así mismo, disminuir el número de registros que usa cada *kernel* supondría una mayor ocupación de los multiprocesadores, dando como resultado una mejora en el rendimiento. En concreto, en [15] han sido capaces de reducirlo hasta 16, obteniendo una ocupación del 67 %. También tenemos que analizar cómo influyen los paquetes y la tasa de divergencia de éstos, y si el uso de paquetes es, en definitiva, perjudicial o no.

Aparte de estas mejoras en la implementación, resulta interesante estudiar el efecto que tienen algunas decisiones sobre el diseño del algoritmo. Por ejemplo, un análisis del rendimiento de otros volúmenes delimitadores, como AABBs, resulta imprescindibles.

Respecto a la forma de construcción de la BVH, habría que probar diferentes órdenes de inserción de triángulos, tomando tiempos para los árboles generados, e incluso implementar otros algoritmos de construcción. Además, habría que evaluar distintos algoritmos de *shading* y comprobar todo lo anterior sobre las escenas de *benchmark* de [21].

Referencias

[1] T. Whitted. An Improved Model for Shaded Display. Graphics and Image Processing. Volumen 23, número 6, 1980.

[2] I. Ward, P. Slusallek, C. Benthin, M. Wagner. Interactive Rendering with Coherent Ray Tracing. Eurographics 2001. Volume 20, number 3, 2001

[3] NVIDIA CUDA Computer Unified Device Architecture, Programming Guide, Version 2.0.

[4] GPGPU, General-purpose computation using graphics hardware, `http://www.gpgpu.org`. última visita, 12 de Agosto de 2008.

[5] F. Mandl. Ray Tracing on Modern Graphics Hardware.

Master Thesis 2005.

[6] F. Karlsson, C. J. Ljungstedt. Ray Tracing fully implemented on programmable graphics hardware. Master's Thesis. 2004.

[7] M. Christen. Ray Tracing en GPU. Master's Thesis. 2005.

[8] N. A. Carr, J. D. Hall, J. C. Hart. The Ray Engine. Graphics hardware 2002, pp 1–10.

[9] T. Purcell. Ray Tracing on a Stream Processor. PhD Thesis 2004.

[10] J. Amanatides, A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing.

[11] V. Havran. Heuristic Ray Shooting Algorithms. PhD Thesis. 2000.

[12] T. Foley, J. Sugerman. KD-Tree Acceleration Structures for a GPU Raytracer. Graphics Hardware (2005)

[13] S. Popov, J. Günther, H. P. Seidel, P. Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracer. Eurographics 2007.

[14] N. Thrane, L. O. Simonsen. A Comparison of Acceleration Structures for GPU Assisted Ray Tracing Master Thesis 2005.

[15] J. Günther, S. Popov, H. P. Seidel, P. Slusallek. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. Eurographics Symposium on Interactive Ray Tracing 2007.

[16] I. Ward. Realtime Ray Tracing and Interactive Global Illumination. PhD Thesis. 2004.

[17] J. Goldsmith, J. Salmon Automatic Creation of Object Hierarchies for Ray Tracing. IEEE CG&A 1987.

[18] T. Möller, B. Trumbore. Fast, Minimum Storage Ray/Triangle Intersection. Journal of Graphics Tools, volumen 2, número 1, páginas 21-28, 1997.

[19] SCAN `http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/scan/doc/scan.pdf`

[20] The Stanford 3D Scanning Repository. `http://graphics.stanford.edu/data/3Dscanrep/`

[21] J. Lext, U. Assarsson. T. Möller. A Benchmark for Animated Ray Tracing. IEEE Computer Graphics and Aplication. 2001.

TRAVERSING A BVH CUT TO EXPLOIT RAY COHERENCE

R. Torres[1] P. J. Martín[2] A. Gavilanes[3]
Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Madrid, Spain.
[1]*r.torres@fdi.ucm.es,* [2]*pjmartin@sip.ucm.es,* [3]*agav@sip.ucm.es*

Keywords: Coherence, Bounding Volume Hierarchy, CUDA, Path Tracing, Ray Tracing.

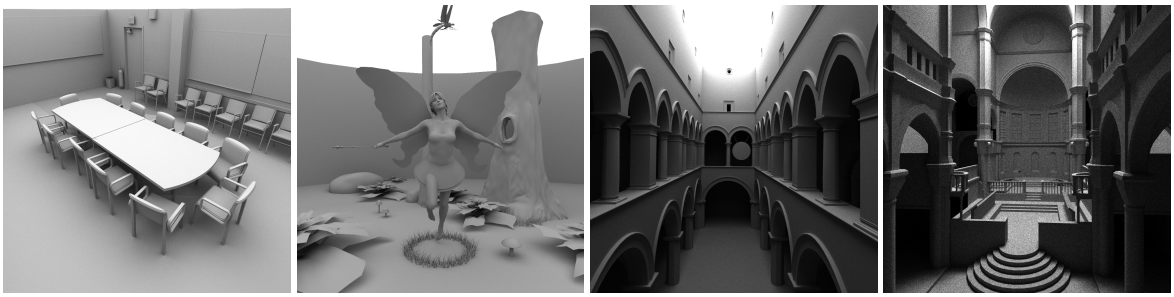
Abstract: In this paper we study how to deal with the ray incoherence that naturally arises in path tracing-based systems. We introduce the notion of BVH *Cut* to split the tree into a forest of disjoint subtrees. We will use it to filter the rays that are successively generated by the path tracing algorithm. Each subtree is then traversed by its corresponding group of rays. Despite the overload of filtering all the rays each time, a significant profit is achieved. Nevertheless, constructing a BVH cut is a challenging task, because it can lead to a huge amount of work if the same rays belongs to many groups. Thus, we present two kind of building heuristics: *structural heuristics* that characterizes the root of a subtree by a property (the node's depth or the surface area of its bounding volume in this paper), and *optimization heuristics* that are based on the Simulated Annealing method. The performance of traversing the cuts so built has been experimentally analyzed over four usual scenes, using two popular implementations of the subtree traversal (*persistent while-while* / *persistent packet*). The results show a relevant saving time w.r.t. the classic BVH traversal, that grows as the ray incoherence increases. The best saving ranges from 32.0% / 40.9% for structural heuristics, to 32.0% / 51.7% for cuts built with Simulated Annealing.

1 INTRODUCTION

One of the main bottlenecks for most ray tracing algorithms is the *traversal* stage. Although great progress has been made in their performance through the usage of modern GPU architectures (Aila and Laine, 2009), the success of efficiently traversing a great amount of incoherent rays in parallel remains a challenging topic, since it is highly connected to the programming SIMD model of the hardware, and, more precisely, to the way rays are arranged on the device. Therefore, the notion of *coherence* is essential to understand the behavior of SIMD-based implementations, and more research on coherence is required to design faster traversal procedures.

Although many definitions of coherence can be found in the literature, most of them refer to a qualitative measure. Thereby, two rays are said to be coherent if they traverse the same nodes and triangles most of the time. In order to exploit coherence in a GPU, rays are usually grouped into *packets*, mainly to allow the rays inside a packet to cooperate when reading scene information from global memory. Thus, ray packets become the traversal logical unit, which gives rise to the so called packet-based traversals. Their main disadvantage is that the rays inside a packet are forced to traverse the hierarchy in the order the packet chooses, which usually increases the total number of nodes traversed w.r.t. the single-ray traversal. Hence, the success of any packet-based traversal leans on the assumption that the saving due to the cooperative reading is greater than this traversal penalty. Consequently, its success depends on the coherence inside each packet, since the more coherent the rays of a packet are, the higher the saving is.

Recently, (Aila and Laine, 2009) suggest that the assumption is not valid for primary, one-bound diffuse and ambient occlusion rays on modern GPUs. Specifically, their experiments show that a stack-based single-ray traversal is faster than a stack-based packet traversal. Nevertheless, although the rays are not grouped in explicit packets, they are implicitly



(a) (b) (c) (d)

Figure 1: Scenes used for our testings. The images have been generated at a resolution of 1024×1024 with 1000 paths per pixel (including primary rays). Each path is formed by 10 rays (each primary ray bounces 9 times). The total rendering times in SINGLE (*persistent while-while*) with Cut_{root} are CONFERENCEROOM=552.7s (a), FAIRYFOREST=556.3s (b), SPONZA=787.1s (c) and SIBENIK=815.4s (d). The total rendering time with the best structural depth Cuts in Table 1 and inter-BVH pruning are CONFERENCEROOM=535.9s (3.03%), FAIRYFOREST=511.4s (8.07%), SPONZA=645.3s (18.01%) and SIBENIK=605.9s (25.69%). The percentage in brackets are the saving w.r.t. Cut_{root}.

grouped since the SIMD model of GPUs is based on the notion of *warp*. Therefore, the single-ray traversal they compute can be actually considered packet-based, since rays are arranged in the warps according to a specific order (Z-order) of the image.

An important inconvenient of most coherence definitions is that it cannot be known before traversing the tree. Thus, heuristics have to be used for packing rays in order to obtain a high coherent level afterward. In the literature, we can find two heuristics. On the one hand, coherence usually has a *geometric meaning*: two rays are said to be *geometrically coherent* whenever their origins lay "near" and/or the angle between their directions is "small" enough. Therefore, the geometric coherent attempts to ensure a deeper coherence, because it is expected for two geometrically coherent rays to traverse the same nodes of the acceleration structure.

So, it is also natural to suggest a *behavioral meaning*: two rays are *behaviorally coherent* w.r.t. a node n of the acceleration structure whenever both rays intersect the bounding volume enclosing n. The underlying idea behind behavioral coherence is that the acceleration structure drives the traversal for all the rays, or an enough big set of the rays, simultaneously. In fact, when a node is explored, only those rays intersecting its bounding volume are considered and the rest of them have to be filtered. In that sense, parallel GPU primitives, such as sorting, compact and (segmented) scan functions, become essential for implementing many tasks during the ray classification. Notice that the success of traversal then depends on the performance of these primitives, and that, although most of them are well known, their effective implementations on GPUs are relatively recent.

In this paper we research how to exploit the behavioral coherence when a great amount of incoherent rays are shot through the scene, which is usual for *path tracing*-based systems. Our main contribution is double. On the one hand, we propose a BVH traversal that begins classifying the rays on GPU according to a sequence of descendants of the root, which will be called *Cut* along the paper. This can be considered a breadth-first traversal for exploiting behavioral coherence, since it results in a set of traversal tasks involving behaviorally-coherent packets. Then, these tasks are finally traversed in a classic depth-first way on GPU. It is worth to mention that our approach does not depend on the implementation of the traversal that it is integrated into the system, since they are fully interchangeable. We have actually tested two of the fastest implementations on GPU –the *persistent packet* and the *persistent while-while* by (Aila and Laine, 2009)– yielding successful saving rates in both cases.

On the other hand, we present different criteria for building cuts that are compared each other regarding the performance of their traversal over four usual scenes. The results show a relevant saving time w.r.t. the classic BVH traversal, that grows as the ray incoherence increases.

2 RELATED WORK

Ray packets. (Wald et al., 2001) are pioneers in using ray packets for developing an interactive ray tracer on CPU. They use the trivial geometric coherence of neighboring pixels to pack primary rays. Packets allow to decrease memory traffic and improve the cache efficiency by exploiting the 4-wide SIMD units.

Later, packets were adapted to the 32-wide SIMD

units on GPUs. Two different directions have been followed in order to simulate on GPU the recursive nature of hierarchical traversals. The first one is based on stacks, which are implemented on shared memory. Some of the papers included in this trend are (Günther et al., 2007) for BVHs, and (Horn et al., 2007) for KD-trees. The second approach introduces new links in the tree to guide its traversal. Examples of these stackless tracers are (Popov et al., 2007) and (Foley and Sugerman, 2005) for KD-trees, and (Torres et al., 2009) for BVHs. Recently, (Zlatuska and Havran, 2010) make a comparison of several GPU implementations, including proposals of both tendencies.

Concerning the efficiency of explicit packets, (Aila and Laine, 2009) question their practical interest. Indeed, their paper shows that traversing each ray independently is faster than traversing ray packets. Nevertheless, it only considers primary and secondary rays, which are arranged on the device according to the image Z-order, thus they are implicitly packed into geometrically coherent warps.

Geometric coherence. The notion of geometric coherence appears very often in the ray tracing bibliography (Wald and Slusallek, 2001), and more especially in those papers concerning packet-based traversals. Thus, we only mention recent papers that analyze different techniques for exploiting geometric coherence, among their main contributions. (Mansson et al., 2007) present several geometric heuristics to organize newly spawn rays. Unfortunately, classifying secondary rays on CPU takes too much time to make them applicable. (Noguera et al., 2009) present a KD-tree traversal for ray packets, using CPU's SEE. Rays are simply classified according to the signs of their directions. (Boulos et al., 2007) propose several ways of packing secondary rays. It shows a performance of around 3x for the method that groups rays of the same type vs. the single ray method.

Behavioral coherence on CPU. Most of the papers concerning behavioral coherence can be classified in two groups. The first one is composed of those works that use the acceleration structure as a reference to pack the rays into coherent packets. Among them, (Pharr et al., 1997) describe a Monte Carlo renderer that takes advantage of the cache units to reduce memory traffic from disk. Furthermore, the rays get enqueued in the voxels of a uniform grid. The *scheduler* subsystem is then responsible for starting the intersection test of the rays in a queue against the geometry at the corresponding voxel, depending on the information already cached.

Similarly, (Navratil et al., 2007) present another technique to decrease the traffic between DRAM and cache L2. Queues are now located at some nodes of a KD-tree, called *queue points*. The subtrees related to these nodes fit in cache L2, which is used to accelerate the traversal of the rays in the queue along the subtree.

More recently, (Boulos et al., 2008) introduce the quantitative notion of SIMD-coherence to measure the utilization of the SIMD units. Specifically, it computes the ratio between the number of active rays and the packet size –which is fixed to 256 rays– to express how coherent the packet is. It then uses filtering techniques to compact those packets whose ratio drops below a threshold. This demand-driven reordering method gives the best results for diffuse path tracing vs. glossy and perfect specular ray tracing.

The second group includes packing techniques based on the operations that the rays demand, instead of the nodes of the structure they pass through. The aim of these proposals is to get the maximum of the SIMD units. Thereby, (Wald et al., 2007) and (Gribble and Ramani, 2008) present ray tracers in which the rays are filtered to output those requiring the same operations. Then, these operations are run over the corresponding rays in a SIMD manner. The experiments included in the former show a high SIMD utilization, for ray streams of 64×64 at most. The performance of the latter is predicted to 6-16 FPS, which is subsequently improved to 15-32 FPS by separating address and data processing (Ramani et al., 2009). In both papers, the size of the ray streams is also up to 64×64.

Behavioral coherence on GPU. (Garanzha and Loop, 2010) is the first paper in explicitly exploiting the notion of behavioral coherence on GPU, as we are concerned. It firstly packs the rays using a geometrical criterion that is based on the direction and the origin of the ray. In order to accelerate the classification, the rays are previously transformed into hashing keys, and then sorted by using fast GPU primitives (Harris et al., CUDPP). Then, the frustum of each packet traverses a BVH in breadth-first order. Finally, a list of leaves is obtained per ray. The rays related to each leaf are then split into packets and tested for intersection with the bounding volume of the leaf and its triangles.

Finally, (Aila and Karras, 2010) present an architecture similar to NVidia Fermi, that reduces the memory traffic between DRAM and on-chip caches. Its traversal is based on hierarchically located queue points in the spirit of (Navratil et al., 2007).

3 BVH CUTS

A *Cut* of a BVH is a set of nodes $C = \{n_1, n_2, \ldots, n_N \,|\, n_i \in \text{BVH}\}$ such that for every leaf l

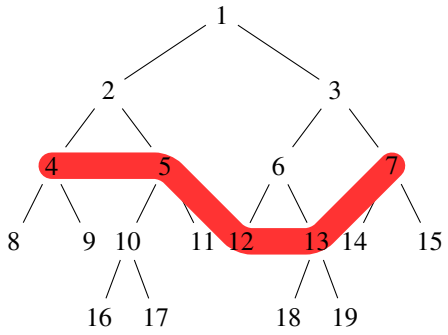


Figure 2: Example of a BVH Cut.

```
1  in:  Cut C;        Ray R[N_R];
2  out: float t^g_hit[N_R];
3  var:
4    float t_hit[N_R]; bool mask[N_R];
5    int id[N_R];      int max_R;

7  for each r ∈ [1..N_R] in parallel do
8    t^g_hit[r] = ∞;

10 // For each node in the Cut
11 for each n_i ∈ C do {
12   // Intersection of all rays
13   // with the BV(n_i) on GPU
14   for each r ∈ [1..N_R] in parallel do {
15     t_hit[r] = ∞;
16     mask[r] = test(r, BV(n_i));
17   }
18   // Compacting on GPU
19   compact(mask, id, max_R);
20   // Traversal on GPU
21   traversal(R, id, max_R, B_i, t_hit, t^g_hit);
22 }
```

Figure 3: Traversing a BVH cut.

in the BVH, there exists a unique node $n_j \in C$ satisfying $l \in subtree(n_j)$ (see Figure 2 for an example). Thereby, a cut partitions the BVH into two disjoint sets of nodes T (top) and B (bottom), with $root \in T$ and all the leaves belong to B –which is actually a forest of N subtrees.

In order to exploit the behavioral coherence, rays are classified into N sets of rays, one per node of the cut. Specifically, a ray r is inserted into the set s_i related to the node n_i, whenever r intersects the bounding volume $BV(n_i)$ of n_i. Observe that a ray can belong to different sets, thus, it can require the subsequent traversal of different subtrees. The classification process can be compared to a breadth-first traversal, since each ray spreads many tasks that are not solved immediately, but later on. Finally, each set s_i is split into packets that are behavioral coherent w.r.t. the node n_i. This splitting is trivial: a set of 32 consecutively rays are arranged into a packet. Moreover, if the set s_i yields a packet p, p is then related to the BVH hanging from the node n_i, which we will call B_i.

Cut Traversing. Figure 3 shows the traversal scheme for a BVH cut. It is mainly composed of three stages. In the first one (lines 14–17) the array *mask* is updated with the intersection test of each ray with $BV(n_i)$. The second stage (function *compact* at line 19) removes the rays that did not pass the last intersection test by compacting the remaining ones. The array *id* stores the indices of the rays that passed the test and max_R keeps the number of them. The third stage (line 21) is a traversal algorithm of the BVH B_i in a depth-first style. Any traversal algorithm is possible in this stage and we have tested two GPU approaches as we will detail in Section 5. The extraction order of the rays to be traversed respects the order inside the array R.

Traversing a cut leads to N classic traversals that compute the nearest intersection point for each ray, inside the part of scene the corresponding B_i covers. As usual, we use distances to refer to points, and thus we write $t_{hit}[r]$ to denote the intersection point related to the *(local)* traversal of the current B_i w.r.t. a given ray r. Notice that these local traversals are run on GPU, but sequentially launched from CPU. Therefore, the final *(global)* distance for r, $t^g_{hit}[r]$, is computed as the minimum among the values $t_{hit}[r]$ related to each B_i.

Regarding the integration of pruning techniques, two improvements can be considered. First, an intra-B_i pruning can be applied, and indeed is applied, when launching the function *traversal* at line 21. The current $t_{hit}[r]$ is then used during the traversal of B_i to rule out farther intersected nodes for r inside B_i.

Second, an inter-B_i pruning could be incorporated at line 15 to suitably initialize the array t_{hit} to the current t^g_{hit}, instead of ∞. Thus, this line would become $t_{hit}[r] = t^g_{hit}[r]$. Again, the aim would be to take advantage of the traversals that have been completed before running the i-th iteration, i.e. the traversals of those B_j with $j < i$. Specifically, B_i could be ruled out if the current $t^g_{hit}[r]$ was less than the entry distance to $BV(n_i)$. Nevertheless, the order among the B_i that leads to the best overall performance cannot be determined in advance. So, we have not implemented this inter-B_i pruning and the results (Section 6) are an upper bound, regardless how the B_i are sorted.

4 CUT CREATION

In order to boost the efficiency of a cut, we must compare the benefit from the behavioral coherence of

```
Cut create_depth(node n, int d) {
  if(isLeaf(n) ∨ depth(n) == d)
    return {n};
  else {
    Cut C_L = create_depth(left(n), d);
    Cut C_R = create_depth(right(n), d);
    return C_L ∪ C_R;
} }
```

```
Cut create_area(node n, float a) {
  if(isLeaf(n) ∨ area(n) < a)
    return {n};
  else {
    Cut C_L = create_area(left(n), a);
    Cut C_R = create_area(right(n), a);
    return C_L ∪ C_R;
} }
```

Figure 4: Implementation of the structural heuristics for building a Cut. Top: cut creation by DEPTH. Bottom: cut creation by AREA.

each packet to the *overload* due to the total number of packet traversals the rays produce. Since both issues are opposite, let us first analyze the two extremes.

The first one corresponds to the case in which the cut is composed of the leaves of the BVH (C_{leaves}). The overload is then more expensive than the benefit, because too many traversals arise: each ray requires a test against each leaf whose bounding volume the ray intersects. Hence, traversing the cut would degenerate into the inefficient brute force.

In the other extreme, the cut is just composed of the root of the BVH (C_{root}). Each ray then traverses the whole BVH from its root in a depth-first way. Thus, the usage of the cut is useless. To sum up, our traversal method is not efficient in both extremes, and a trade-off between the benefit and the overload of using a BVH cut should be found. Hence, we present two different group of heuristics for building cuts, which are later compared with respect to their performance over usual scenes.

4.1 Structural Heuristics

The first group corresponds to *structural* heuristics, because the resulting cuts are composed of those nodes satisfying certain property that only depends on the structure of the BVH. In our experiments we have tested two properties that are respectively based on the node's *depth* (called DEPTH), and on the *surface area* of its bounding volume (called AREA). Concretely, the cuts consist of the nodes at a given depth d for the DEPTH heuristic, while it is composed of the

```
Cut Simulated_Annealing(node root){
 Cut currentCut = {root};
 float currentTime = render(currentCut);
 Cut bestCut = currentCut;
 float bestTime = currentTime;
 Cut nextCut = evolve(currentCut);
 float nextTime = render(nextCut);
 int temp = MAX_TEMP;

 for(i=0; i< NSteps; i++){
  for(j=0; j< NSteps_per_Temp; j++){
    // Acceptance threshold
    float p = exp(|currentTime−nextTime| / temp);
    if((nextTime < currentTime) ∨ (rand(0,1) < p)){
      currentCut = nextCut;
      currentTime = nextTime;
      // Update the bestTime
      if(currentTime < bestTime) {
        bestTime = currentTime;
        bestCut = currentCut;
      }
    }

    nextCut = evolve(currentCut);
    nextTime = render(nextCut);
  }// for j
  temp = α · temp;
 }// for i

 return bestCut;
}
```

Figure 5: Implementation of Simulated Annealing for building a cut.

first nodes from the root whose surface area falls below a given threshold a for the AREA heuristic.

Figure 4 shows how to build a cut in function of the property. Observe that a leaf l is immediately added to the cut, although the property did not hold for any node in the path from the root to l. This prevents the traversal from ruling out parts of the scene.

4.2 Simulated Annealing

The cut construction can be formulated as an optimization problem. Thus, our second group of heuristics consists of methods that look for the minimum solution inside a search space that is composed of all possible cuts. The objective function to be minimized is the render time a cut traversal requires. According to this formulation, many of the algorithms employed in *combinatorial optimization* can be used to find the best cut. Nevertheless, searching for the best

cut turns to be unfeasible, as it usually happens for many combinatorial optimization problems, hence we focus on approximation algorithms. Among the existing algorithms, we have adapted the *Simulated Annealing* method (*SA* in the sequel), since it can be easily applied to these problems, due to its generic nature (Zomaya and Kazman, 1999).

SA can be described as a *randomized iterative improvement algorithm*, since it does not only accept decreasing moves, regarding the given objective function, but it also tolerates increasing moves in order to avoid getting trapped in local minima. Indeed, it uses a probability function, that decreases as the execution advances, for accepting increasing moves. The method asymptotically converges to a global minimum, whenever certain conditions hold, concerning the *annealing schedule*.

Figure 5 describes how to build a BVH cut using SA. Besides the current cut (*currentCut*), the algorithm also holds another one (*nextCut*) that corresponds to a random evolution of the former. These two cuts advance together along the execution of two nested loops: one for decreasing the control parameter $temp$ (line 10) –the *temperature* used in the original SA formulation– and another one for trying many moves at the same $temp$ (line 11). Regarding increasing the render time, the algorithm accepts those cuts whose acceptance threshold (line 13) is greater than a uniform random value in [0,1] (line 14). If the *nextCut* is finally accepted, it is assigned to *currentCut* (line 15) and the best cut is updated if required (lines 18–21). In any case, a new random evolution is computed (line 24) and subsequently stored in *nextCut*.

The function *evolve* generates a reachable cut from *currentCut* by applying either the *join* or the $unfold$ operation. In the former, an inner node n of the cut C is replaced by its two children: $unfold(C,n) = (C - \{n\}) \cup \{left(n), right(n)\}$, whereas two sibling nodes $l, r \in C$ of C are replaced by their father in the later: $join(C,l,r) = (C - \{l,r\}) \cup \{father(l)\}$ In this function, one of these operations is randomly chosen (if both are possible).

5 EXPERIMENTAL SETTINGS

Our application has been run on a NVIDIA GeForce GTX 285 with 1GB of RAM. The test scenes are FAIRYFOREST, CONFERENCEROOM, SPONZA and SIBENIK (see Figure 1). The FAIRYFOREST scene is open but a quadrilateral has been positioned as a roof, preventing the rays from escaping from the scene. All the images have been taken at a resolution of 1024×1024.

The BVHs have been built by following the Surface Area Heuristics (SAH) by (Goldsmith and Salmon, 1987) and using the greedy top-down algorithm by (Ize et al., 2007). To improve the overall performance of the BVH, we have also applied the *early split clipping* technique by (Ernst and Greiner, 2007). So, before starting the construction, the bounding volume of each triangle is iteratively halved until its surface area is lower than a certain threshold.

We have used *path tracing* (Kajiya, 1986) as our ray tracing algorithm, and for the sake of convenience, every surface of the scene is considered as diffuse (i.e. with a constant BRDF). Hence, as soon as a ray finds the nearest intersection point, a new ray is spawned. Its origin is the intersection point and its direction is randomly chosen over a virtual hemisphere on the surface normal. We have considered the cosine as the probability density function, i.e. those points near the pole have more probability because it depends on $cos\,\theta$ (where θ is the angular deviation of the point from the pole). Since the number of rays does not increase, we have an absolute control over the memory that is actually allocated.

Each ray is bound to a *persistent* CUDA thread, according to (Aila and Laine, 2009). The set of rays whose associated threads are simultaneously launched is called a *generation*. Generations are enumerated; the generation 0 is composed of the primary rays, and the generation i is composed of the rays spawn from the generation $i-1$. The number of considered generations in this paper is fixed to 10. The number of rays in a generation is the biggest one that our implementation and our graphics card are able to store: 8 MRays ($= 8 \cdot 2^{20}$ rays). The primary rays are spawned from a bidimensional array of 4096×2048. Since the images are at a resolution of 1024×1024, each subarray of 4×2 rays contains 8 samples for the same pixel. When it is stored in memory, the bidimensional array is flattened according to the Z-order (Morton code).

In these settings, path tracing is specially suitable for our experiments since no property can be assumed in advance for the rays from generation 0 on (i.e. no primary rays). As we will see in Section 6, the incoherency becomes maximal from generation 2 on.

We have used the linear congruential generator by (Park and Miller, 1988) as random number generator algorithm. It has a period of $2^{31} - 2$, which is greater than the total amount of random numbers needed in the tests, ensuring that each ray receives different random numbers.

Our path tracer has been implemented with five CUDA kernels: *RayGenerator (RG)*, *Test*, *Compact*,

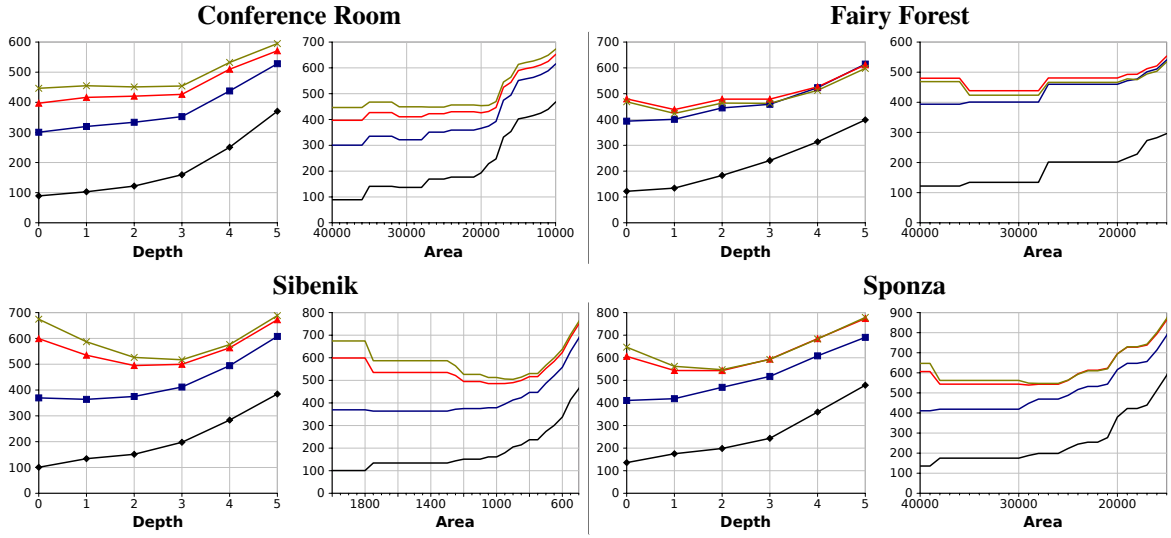


Figure 6: Render times (in ms) measured for the four scenes with the traversal algorithm SINGLE by using structural cuts. The colors are: black (generation 0), blue (generation 1), red (generation 2), green (generation 3).

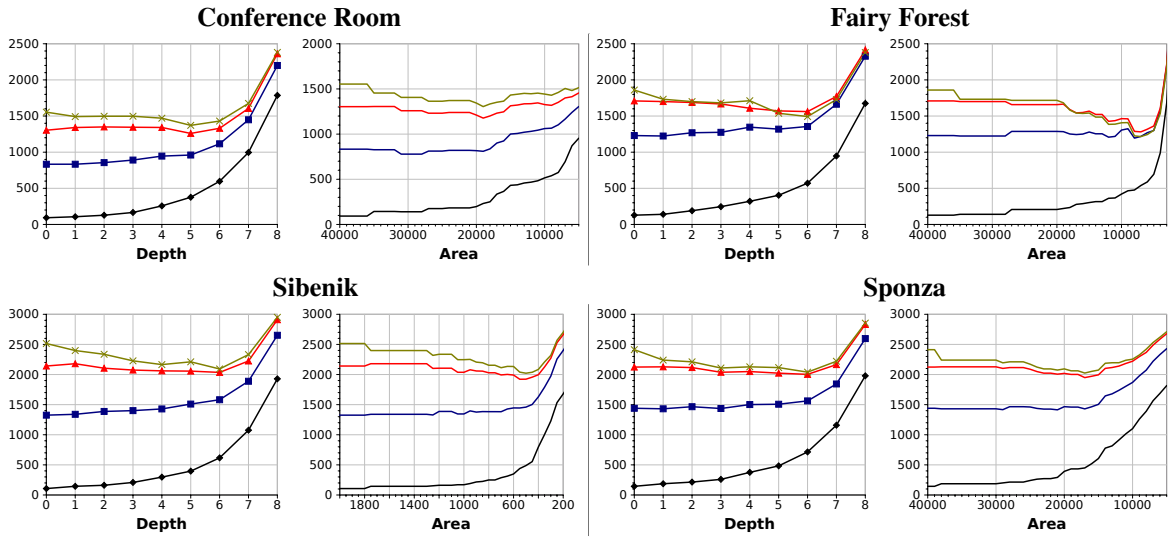


Figure 7: Render times (in ms) measured for the four scenes with the traversal algorithm PACKET by using structural cuts. The colors are: black (generation 0), blue (generation 1), red (generation 2), green (generation 3).

TraversalIntersection (TI) and *Shader (SH)*. The algorithm runs according to the following scheme. First, the primary rays are spawned from a pinhole camera, in the kernel *RG*. Then, in the kernel *Test*, the rays are tested for intersection with a node n of the cut. Next, the rays that passed the previous intersection test are compacted, in the kernel *Compact*. This kernel is actually the primitive *cudppCompact* of the CUDPP library by (Harris et al., CUDPP) and preserves the Z-order of the initial rays. Afterward, the kernel *TI* finds the nearest intersection for every ray by traversing the subtree hanging from n. The two algorithms used for traversing a subtree are due to (Aila and Laine, 2009). They are the *persistent packet* and the *persistent while-while* and will be denoted by PACKET and SINGLE, respectively. Finally, a new secondary ray is spawned over the hemisphere from the nearest intersection in the kernel *SH*.

Table 1: The percentage of saving in render time of the best cut built with the DEPTH heuristics w.r.t. C_{root}. The numbers in brackets are the depths of the best cuts.

SINGLE										
Scene \ Gen.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Conf.Room	0.0(0)	0.0(0)	0.0(0)	0.0(0)	0.1(2)	1.4(2)	1.6(2)	2.0(2)	2.1(2)	2.1(2)
FairyForest	0.0(0)	0.0(0)	8.6(1)	9.6(1)	10.0(1)	9.7(1)	9.5(1)	9.4(1)	9.3(1)	9.0(1)
Sibenik	0.0(0)	1.5(1)	17.3(2)	23.2(3)	26.2(3)	28.0(3)	29.0(3)	29.8(3)	30.3(3)	**30.6(3)**
Sponza	0.0(0)	0.0(0)	10.4(2)	15.4(2)	17.1(2)	18.3(2)	19.0(2)	19.5(2)	19.8(2)	20.1(2)
PACKET										
Scene \ Gen.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Conf.Room	0.0(0)	0.0(0)	3.3(5)	11.7(5)	7.4(5)	14.3(5)	9.0(5)	14.4(5)	9.0(5)	14.4(5)
FairyForest	0.0(0)	0.5(1)	8.6(6)	19.7(6)	16.7(6)	22.1(6)	18.0(6)	**22.7(6)**	18.1(6)	22.6(6)
Sibenik	0.0(0)	0.0(0)	4.9(6)	16.7(6)	13.8(6)	20.1(6)	17.0(6)	22.3(6)	17.8(6)	21.8(6)
Sponza	0.0(0)	0.6(1)	5.7(6)	15.4(6)	12.8(6)	17.9(6)	14.8(6)	19.7(6)	15.6(6)	20.0(6)

Table 2: The percentage of saving in render time of the best cut built with the AREA heuristics w.r.t. C_{root}. The numbers in brackets are the percentage of surface area related to the best cut w.r.t. the surface area of the root.

SINGLE										
Scene \ Gen.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Conf.Room	0.0(100)	0.0(100)	0.0(100)	0.0(100)	0.8(75.3)	2.2(75.3)	2.4(75.3)	2.7(75.3)	2.4(75.3)	2.3(75.3)
FairyForest	0.0(100)	0.0(100)	8.6(99.4)	9.6(99.4)	10.0(99.4)	9.7(99.4)	9.5(99.4)	9.4(99.4)	9.3(99.4)	9.0(99.4)
Sibenik	0.0(100)	1.5(99.5)	18.9(59.7)	25.1(51.2)	27.9(51.2)	29.6(51.2)	30.6(51.2)	31.3(51.2)	31.7(51.2)	**32.0(51.2)**
Sponza	0.0(100)	0.0(100)	11.1(75.3)	15.4(72.7)	17.1(72.7)	18.3(72.7)	19.0(72.7)	19.5(72.7)	19.8(72.7)	20.1(72.7)
PACKET										
Scene \ Gen.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Conf.Room	0.0(100)	6.5(86.4)	9.7(53.0)	16.0(53.0)	10.5(53.0)	16.6(53.0)	10.7(53.0)	16.3(53.0)	10.5(53.0)	15.7(53.0)
FairyForest	0.0(100)	2.8(22.7)	25.0(19.8)	34.6(19.8)	34.3(19.8)	39.1(19.8)	36.4(19.8)	40.3(19.8)	37.1(22.7)	**40.9(22.7)**
Sibenik	0.0(100)	0.0(100)	10.2(31.2)	19.6(28.4)	17.6(28.4)	23.5(31.2)	20.0(28.4)	25.1(28.4)	20.8(28.4)	24.9(28.4)
Sponza	0.0(100)	1.7(54.5)	8.2(44.1)	16.0(44.1)	13.6(44.1)	19.7(44.1)	15.4(44.1)	20.0(44.1)	16.8(44.1)	19.7(44.1)

6 RESULTS

Structural Heuristics. Several structural cuts have been built with different values for the parameter of the DEPTH and AREA heuristics. The render time for their traversal are depicted in Figure 6 for SINGLE and in Figure 7 for PACKET. In the *y*-axis, the measured render times (in *ms*) of the cut traversal are displayed. In the *x*-axis, different values of the parameter are included. Points of the same generation are joined in a continuous line. However, only the first four generations are showed for the sake of clarity, which gives rise four curves per chart. The remaining ones have a behaviour similar to generation 3.

The first (the leftmost) value of the parameter always corresponds to the structural value that builds C_{root}. Therefore, the first value of each curve corresponds to the SINGLE or PACKET traversal of the whole BVH plus an extra time due to filtering (around 10 ms according to our measures). Higher values in DEPTH and lower values in AREA provoke an exponential growth in render time, which is not included in the charts. We have measured generations for different random number seeds. The results are very similar and only the charts for one seed are displayed on the figures.

As it can be seen, the curves of a given generation have a similar shape in every scene. The curves of generation 0 (primary rays) and generation 1 do not undergo any improvement w.r.t. the traversal of C_{root}. On the contrary, the generations 2 to 9 have a drop at the beginning and an exponential increase after. The depth of this valley depends both on the scene as well as on the traversal algorithm.

The valley is deeper for PACKET than for SINGLE. As (Aila and Laine, 2009) mention, SINGLE is more efficient than PACKET for coherent (such as primary rays) and non-coherent rays. This is due to the fact that the memory bandwidth in modern GPUs is high, and the bottleneck in PACKET is not the memory traffic but the additional amount of traversed nodes.

Notice that, the minimum of each curve occurs more to the left in SINGLE than in PACKET (i.e. in shallower nodes or with bigger surface area). The overload in both algorithms is the same, so the memory system must be the responsible for this difference.

Table 3: The percentage of saving in render time of the best cut found with Simulated Annealing w.r.t. C_{root}. The numbers in brackets (D/A) are: D, the averaged depth of the nodes in the cut; and A, the percentage of averaged surface area of the nodes in the cut w.r.t. the surface area of the root.

SINGLE										
Scene \ Gen.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Conf.Room	0.0	0.0	0.0	2.5	3.9	5.3	4.9	5.3	5.0	5.0
	(0.0/100)	(0.0/100)	(0.0/100)	(4.4/46.9)	(4.5/43.2)	(4.6/42.9)	(4.0/49.0)	(4.7/42.4)	(4.1/48.8)	(4.8/42.2)
FairyForest	0.0	5.9	17.1	16.3	15.8	14.7	14.0	13.4	12.8	12.4
	(0.0/100)	(5.0/17.9)	(5.1/26.1)	(4.4/31.2)	(4.4/31.2)	(4.4/31.2)	(4.4/31.2)	(4.4/31.2)	(4.4/31.2)	(4.4/31.2)
Sibenik	0.0	1.5	18.9	25.4	28.0	29.6	30.6	31.3	31.7	**32.0**
	(0.0/100)	(1.0/71.3)	(2.8/46.9)	(3.0/45.5)	(3.0/45.5)	(3.1/43.8)	(3.1/43.8)	(3.1/43.8)	(3.1/43.8)	**(3.1/43.8)**
Sponza	0.0	0.0	11.1	15.4	17.1	18.3	19.0	19.5	19.8	20.1
	(0.0/100)	(0.0/100)	(1.6/68.2)	(2.0/64.5)	(2.0/64.5)	(2.0/64.5)	(2.0/64.5)	(2.0/64.5)	(2.0/64.5)	(2.0/64.5)
PACKET										
Scene \ Gen.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Conf.Room	0.0	21.8	31.5	37.0	33.3	37.2	32.1	36.0	30.7	34.5
	(0.0/100)	(6.1/26.8)	(6.6/18.3)	(6.6/17.4)	(6.5/17.0)	(6.5/17.4)	(6.5/17.0)	(6.5/17.4)	(6.5/17.4)	(6.4/17.7)
FairyForest	0.0	45.1	49.4	**51.7**	48.5	51.4	47.9	47.7	47.4	47.6
	(0.0/100)	(5.7/23.5)	(5.8/20.0)	**(5.9/17.9)**	(5.8/16.6)	(5.9/17.9)	(5.9/17.8)	(5.7/17.8)	(5.9/17.9)	(5.7/17.8)
Sibenik	0.0	9.3	14.8	21.4	18.8	23.7	20.3	24.5	20.9	24.9
	(0.0/100)	(5.8/28.0)	(6.1/22.4)	(6.2/21.3)	(5.9/20.9)	(6.0/20.9)	(6.0/21.4)	(6.0/21.8)	(5.9/21.5)	(6.0/21.8)
Sponza	0.0	3.4	14.8	24.1	21.5	26.6	22.9	27.3	23.2	27.5
	(0.0/100)	(5.5/32.0)	(6.0/29.3)	(6.4/28.2)	(6.5/30.9)	(6.4/29.9)	(6.3/30.1)	(6.2/30.3)	(6.3/30.1)	(6.3/29.9)

If the packets are more coherent in SINGLE, the number of nodes read from memory does not vary, but the texture caches are better used. On the contrary, if the packets are more coherent in PACKET, the number of nodes read from memory decreases, but the texture cache usage is the same. Therefore, the curves show that the improvement due to the diminishment of the read nodes becomes relevant more to the right than the benefit of cache.

For a given scene, the shape of the curves are very similar in the DEPTH and AREA charts. This fact is not surprising since deeper nodes have also smaller surface areas.

The generations 0 and 1 have not an improvement by the use of cuts. This is due to the fact that these rays are very coherent and the improvement obtained by launching more coherent packets is not enough to exceed the overload.

Tables 1 and 2 summarize the best saving of the figures. They include a column for each generation that shows the percentage of saving of the best structural cut w.r.t. the performance of traversing C_{root}. Hence, it is computed by comparing the first value of the corresponding curve with its minimum, that is, through the expression $\frac{t_{root}-t_{min}}{t_{root}}$, where t_{min} and t_{root} denote these two values. The most relevant savings are 30.6%/32.0% (DEPTH/AREA) for SINGLE applied to SIBENIK, while 22.7%/40.9% for PACKET applied to FAIRYFOREST.

Simulated Annealing. The results can be seen on Table 3. The parameters used are MAX_TEMP=600, NSteps=1000, NSteps_per_temp=1000 and α=0.99.

Observe that the percentage of saving is always better than those related to structural cuts. This is natural since SA manages other cuts apart from structural cuts.

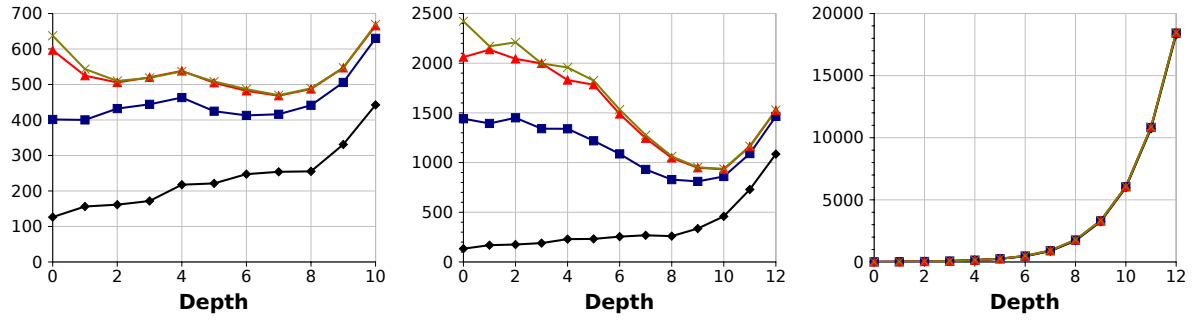
For some scenes, there is a correspondence between the averaged depth of the best SA cut and the best structural-depth cut (e.g. SIBENIK with SINGLE). However, this cannot be generalized to all scenes.

7 DISCUSSION AND FUTURE WORK

The benefit of the usage of cuts is consequence of the fact that the overload due to filtering is less than the improvement obtained by traversing more coherent rays. It is an open issue if this technique is also applicable to CPU ray tracers, other rendering algorithms (such as bidirectional path tracing), other non-diffuse surfaces (such as specular or glossy), and other acceleration structures (such as KD-trees).

Figures 8a and 8b show the render time for only the kernel *TI* concerning SINGLE and PACKET respectively. Observe that the curves of highly incoherent generations (red and green) present a minimum showing that a cut at a certain depth leads to a relevant improvement. Nevertheless, the overload due to filtering grows exponentially (Figure 8c). This is why the minima in Figures 6 and 7 are shifted to the left. It is necessary to study ways of making the most of that coherence or diminishing the overload.

In order to diminish the overload (number of fil-



(a) (b) (c)

Figure 8: (a) render time without overload (only *TI*) for SPONZA and SINGLE; (b) render time without overload (only *TI*) for SPONZA and PACKET; (c) overload for the subfigures (a) and (b).

ters), two cuts $C1$ and $C2$ can be used. The nodes of $C1$ are used to filter the rays whereas the nodes of $C2$ are used to traverse the scene. Each node $n \in C1$ is linked to a set of nodes $\{n'_1, \ldots, n'_N\} \subseteq C2$, such that the nodes n'_i are descendants of n. Thus, the number of filters are fewer than the amount of nodes in $C2$ (since $|C1| \leq |C2|$). The inconvenient is that the rays launched for traversal are more incoherent. We did not obtain successful results and this technique was dismissed.

Nowadays, there already exist cards with more DRAM capacity than the one used in this paper (e.g. the Tesla C2070 has 6 GB). A bigger amount of memory would allow more rays to be stored and traversed in parallel. Thus, the coherence would be higher and better results would be expected. However, this analysis should be experimentally evaluated.

In this paper, the russian roulette method for finishing a path has not been implemented. On the contrary, every ray keeps alive till generation 9. It is expected that high generations will not behave similarly if the size of their populations is different.

The time used to build our cuts are not included in the results, since the construction is considered as a preprocess. It would be worth to study methods that quickly find an effective cut in order to execute the construction during rendering.

8 CONCLUSIONS

In this paper we have studied how to deal with the ray incoherence that naturally arises in path tracing-based systems. In order to improve the BVH traversal of a great amount of incoherent rays, we split the BVH structure into a forest of disjoint subtrees, called Cut, that will be used to group the rays that are successively generated. Each subtree is then traversed by state-of-the-art algorithms: persistent while-while and persistent packet. We experimentally show that, despite the overload of filtering all the rays for each subtree, the subsequent traversal of all these subtrees results faster than traversing the whole BVH. The reason is that the rays traversing a subtree are more coherent according to the behavioral criterion.

We have presented two kinds of heuristics for building a BVH cut. The first one corresponds to structural properties such as the node's depth and the surface area of the bounding volume of the node. For the second one, the construction of the cut is formulated as an optimization problem, and the Simulated Annealing method is applied to build the best cut. Our experiments show that using a cut results in a significant improvement w.r.t the classic traversal of the BVH. Moreover, this improvement increases according to the incoherent measure of the ray generation. The saving depends on the scene, and also on the traversal algorithm (persistent while-while / persistent packet). For example, for the FAIRYFOREST scene, the best saving times for DEPTH are 10.0% / 22.7% (SINGLE / PACKET), for AREA are 10.0% / 40.9%, and for Simulated Annealing are 17.1% / 51.7%.

ACKNOWLEDGEMENTS

This paper has been supported by the Spanish projects CCG10-UCM/TIC-5476 and BSCH-UCM GR58/08-921547.

REFERENCES

Aila, T. and Karras, T. (2010). Architecture considerations for tracing incoherent rays. In *Proceedings of the High-Performance Graphics 2010.*

Aila, T. and Laine, S. (2009). Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High-Performance Graphics 2009*, pages 145–149.

Boulos, S., Edwards, D., Lacewell, J. D., Kniss, J., Kautz, J., Wald, I., and Shirley, P. (2007). Packet-based Whitted and Distribution Ray Tracing. In *Proceedings of Graphics Interface 2007*, pages 177–184.

Boulos, S., Wald, I., and Benthin, C. (2008). Adaptive ray packet reordering. *Symposium on Interactive Ray Tracing*, 0:131–138.

Ernst, M. and Greiner, G. (2007). Early split clipping for bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 73–78.

Foley, T. and Sugerman, J. (2005). KD-tree acceleration structures for a GPU raytracer. In *HWWS'05 Conference on Graphics Hardware*, pages 15–22.

Garanzha, K. and Loop, C. (2010). Fast ray sorting and breadth-first packet traversal for GPU ray tracing. In *Eurographics*, volume 29.

Goldsmith, J. and Salmon, J. (1987). Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Application*, 7(5):14–20.

Gribble, C. P. and Ramani, K. (2008). Coherent ray tracing via stream filtering. In *IEEE/Eurographics Symposium on Interactive Ray Tracing*, pages 59–66.

Günther, J., Popov, S., Seidel, H.-P., and Slusallek, P. (2007). Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the Eurographics Symposium on Interactive Ray Tracing*, pages 113–118.

Harris, M., Owens, J. D., Sengupta, S., Tzeng, S., Zhang, Y., Davidson, A., and Satish, N. CUDA data parallel primitives library (CUDPP). `http://gpgpu.org/developer/cudpp`.

Horn, D. R., Sugerman, J., Mike, H., and Hanrahan, P. (2007). Interactive KD-tree GPU raytracing. In *I3D'07: Proceedings of the symposium on Interactive 3D graphics and games*, pages 167–174.

Ize, T., Wald, I., and Parker, S. G. (2007). Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, pages 101–108.

Kajiya, J. T. (1986). The rendering equation. *SIGGRAPH Computer Graphics*, 20(4):143–150.

Mansson, E., Munkberg, J., and Akenine-Moller, T. (2007). Deep coherent ray tracing. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 79–85.

Navratil, P. A., Fussell, D. S., Lin, C., and Mark, W. R. (2007). Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 95–104.

Noguera, J. M., Ureña, C., and García, R. J. (2009). A vectorized traversal algorithm for ray-tracing. In *International Conference on Computer Graphics Theory and Applications (GRAPP 2009)*, pages 58–63.

Park, S. K. and Miller, K. W. (1988). Random number generator: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201.

Pharr, M., Kolb, C., Gershbein, R., and Hanrahan, P. (1997). Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108.

Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. (2007). Stackless KD-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum (Proceedings of Eurographics)*, 26(3):415–424.

Ramani, K., Gribble, C. P., and Davis, A. (2009). Streamray: A stream filtering architecture for coherent ray tracing. In *Internationa Conference on Architectural Support for Programming Languajes and Operating System*, pages 325–336.

Torres, R., Martín, P. J., and Gavilanes, A. (2009). Ray casting using a roped BVH with CUDA. In *Proc. Spring Conference on Computer Graphics*, pages 107 – 114.

Wald, I., Benthin, C., Wagner, M., and Slusallek, P. (2001). Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of Eurographics'01)*, volume 20, pages 153–164.

Wald, I., Gribble, C. P., Boulos, S., and Kensler, A. (2007). SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Technical Report UUSCI-2007-012.

Wald, I. and Slusallek, P. (2001). State of the art in interactive ray tracing. In *State of the Art Reports, EUROGRAPHICS 2001*, pages 21–42.

Zlatuska, M. and Havran, V. (2010). Ray tracing on a GPU with CUDA – comparative study of three algorithms. In *Proceedings of WSCG'2010, communication papers*, pages 69–76.

Zomaya, A. and Kazman, R. (1999). *Handbook of Algorithms and Theory of Computation*, chapter Simulated Annealing Techniques, pages 37.1–33.19. CRC Press.

CEIG - Spanish Computer Graphics Conference (2012)
Isabel Navazo and Gustavo Patow (Editors)

Generating Coherent Ray Directions in Path Tracing

R. Torres[1] and P. J. Martín[2] and A. Gavilanes[3]

Universidad Complutense de Madrid, Madrid, Spain
[1]r.torres@fdi.ucm.es, [2] pjmartin@sip.ucm.es, [3]agav@sip.ucm.es

Abstract

The quality of the images produced by a global illumination rendering engine is highly connected to the number of paths that are randomly generated from the camera to the light sources. Graphics Processing Units (GPUs) have been used to implement renderers by typically binding each thread to a ray. Nevertheless, this purely random generation does not fit well on the architecture of current GPUs, due to their SIMD nature.
We modify the way paths are extended in order to take the most of GPUs. The arrangement of rays is split into groups of contiguous rays. The hemisphere on each intersection point is divided into spherical patches, and the same patch is chosen for all the rays in the group. The next direction for each path will be randomly chosen on that patch. Thus, if the intersection points of a group are near, the new rays spawned will be similar.
We have implemented different configurations of this idea on a path tracer in CUDA and they have been experimentally tested on usual scenes. In the same amount of time, most of our algorithms complete more paths and, therefore, they produce images of higher quality than those obtained by the purely random generation.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing, I.3.1 [Computer Graphics]: Hardware architecture—Parallel processing

1. Introducción

Simular por ordenador el transporte de luz es necesario para generar imágenes fotorrealistas a partir de escenas tridimensionales. Una de las propuestas para simularlo [Vea98] lo resuelve en términos de la integral

$$I = \int_{\bar{x}\in\Omega} f(\bar{x})\, d\bar{x}$$

donde Ω es el conjunto de todas la rutas de longitud finita que comienzan en una luz y terminan en la cámara, y f es la contribución de una ruta. Una ruta de longitud $n \geq 1$ es una sucesión de puntos $\bar{x} = x_0 x_1 \ldots x_n$, donde x_0 es un punto en la superficie de una luz, x_n está en la lente de la cámara y el resto son puntos que están en la superficie de los objetos de la escena. La contribución f de la ruta $\bar{x}$ es

$$f(\bar{x}) = L_e(x_0,x_1)G(x_0,x_1)\prod_{i=1}^{n-1} f_r(x_{i-1},x_i,x_{i+1})G(x_i,x_{i+1})$$

donde L_e es la *radiance* de emisión de la luz, G es el término geométrico y f_r es la BRDF.

Una aproximación de la integral anterior se puede obtener mediante técnicas numéricas conocidas como Monte Carlo. De esa manera, un estimador de la integral anterior es

$$I \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(\overline{X_i})}{p(\overline{X_i})}$$

donde cada $\overline{X_i}$ es una ruta aleatoria y p es la función de densidad de probabilidades *pdf* de esa ruta.

Uno de los algoritmos usados para generar aleatoriamente rutas es el algoritmo conocido como *path tracing* [Kaj86]. Este algoritmo comienza eligiendo un punto en la lente de la cámara y trazando un rayo hacia la escena. Una vez que se ha encontrado el punto de intersección más cercano para ese rayo, la ruta se extiende eligiendo una nueva dirección aleatoriamente dentro del hemisferio centrado en la normal de la superficie en ese punto. Así se procede hasta que la ruta alcanza una luz o termina mediante ruleta rusa.

Por otro lado, las GPUs han evolucionado hasta convertirse en dispositivos masivamente paralelos con una gran potencia de cálculo. La implementación de un path tracing en GPU puede parecer trivial ya que cada ruta se traza independientemente de las otras. Sin embargo, las GPUs actuales tienen un modelo de ejecución SIMD (SIMT

DOI:10.2312/LocalChapterEvents/CEIG/CEIG12/081-090.

según la terminología de NVidia [NVI11]), lo que significa que existen dependencias entre hilos durante la ejecución. En concreto, peticiones a memoria y divergencias dentro del código pueden obligar a que ciertas operaciones se realicen secuencialmente en vez de en paralelo, lo que implica una disminución del rendimiento.

Según nuestra experiencia, la etapa más lenta del path tracing es la conocida como *traversal*. Esta etapa se encarga de encontrar la intersección más cercana de cada rayo. Si durante el traversal, las direcciones de los rayos son muy *diferentes*, los accesos a memoria estarán muy alejados entre sí. Esto implica un mal uso de las cachés de memoria y un aumento de la cantidad de memoria transferida.

Una forma de solucionar este problema es reordenar estos rayos aleatorios para que aquellos que tengan direcciones similares se ejecuten en hilos adyacentes. En este trabajo vamos a solucionar el problema favoreciendo esto en la medida de lo posible. Los rayos se generan ya agrupados, evitando, por consiguiente, cualquier tipo de sobrecarga relacionada con la ordenación. A esto lo vamos a llamar *generación coherente*. Sin embargo, esta generación conlleva una correlación entre estimadores, lo que puede incrementar el error medio. En este trabajo demostramos que, aunque el error producido por las muestras es mayor, el número de rayos lanzados es suficientemente grande como para que las imágenes resultantes tengan un error menor.

2. Trabajos relacionados

Una de las principales mejoras para implementar ray tracers más rápidos es aprovechar la *coherencia* de un conjunto de rayos. Decimos que un conjunto de rayos es coherente si todos ellos tienen una característica común que beneficia el trazado de estos por la escena. Esta definición, en realidad, depende de los autores que se han ocupado del tema.

Wald et al. [WBWS01] fueron pioneros en el uso de la coherencia para desarrollar un ray tracer interactivo en CPU. En su trabajo, los rayos primarios generados a través de píxeles cercanos se llaman coherentes ya que tienen direcciones parecidas y un origen común. Por tanto, es muy probable que estos rayos recorran los mismos nodos de la estructura de aceleración e intersequen con los mismos triángulos. Para amortizar ciertas operaciones, estos rayos se agrupan en *paquetes*, constituyendo así la nueva unidad de recorrido. El uso de paquetes permite mejorar la eficiencia de las cachés y el uso de las unidades SIMD de las CPUs.

Wald et al. [WKB*02] usan las técnicas de *photon mapping* [Jen96] e *instant radiosity* [Kel97] para implementar un algoritmo de iluminación global en CPU. Los rayos de sombra trazados a los puntos de luz virtuales son también coherentes ya que son generados de manera parecida a los primarios.

Mansson et al. [MMAM07] presentan varias heurísticas geométricas para organizar los nuevos rayos generados en CPU. También en CPU, Noguera et al. [NUG09] presentan un recorrido donde los rayos son clasificados en función del signo de su dirección. Boulos et al. [BEL*07] proponen varias formas de empaquetar rayos secundarios en función de su tipo.

Con la llegada de las GPUs completamente programables, el concepto de paquete de rayos coherentes fue adaptado a las unidades SIMD de anchura 32 de las GPUs. Günther et al. [GPSS07] realizan un recorrido de una BVH usando una pila por paquete, implementada en memoria compartida. Popov et al. [PGSS07] y Horn et al. [HSMH07] proponen recorridos de KD-trees sin pila. Aila y Laine [AL09] demuestran que el recorrido de una BVH usando una pila individual por rayo es más eficiente que el uso de una pila compartida por todo el paquete.

Garanzha y Loop [GL10] empaquetan los rayos usando un criterio geométrico, basado en la dirección y el origen de los rayos. Para acelerar la clasificación, los rayos son previamente transformados en claves *hash* y posteriormente ordenados rápidamente en GPU. Después, el *frustum* de cada paquete se encarga de recorrer la BVH en anchura.

Otra manera de interpretar la coherencia es tener en cuenta la estructura de aceleración durante el recorrido de los rayos. Así, Pharr et al. [PKGH97] describen un renderizador de Monte Carlo usando una rejilla uniforme como estructura de aceleración. Los rayos quedan esperando en los vóxeles de la rejilla, retrasándose la intersección con la geometría contenida. Un planificador se encarga posteriormente de comenzar los tests de intersección de los rayos de una cola contra la geometría de ese vóxel. Este método reduce el tráfico de las cachés de disco. Similarmente, Navratil et al. [NFLM07] presentan otra técnica para decrementar el tráfico entre DRAM y caché L2. En este caso, las colas de rayos están localizadas en algunos nodos del KD-Tree, cuyos subárboles caben en L2. Boulos et al. [BWB08] presentan técnicas de filtrado para compactar aquellos paquetes durante el recorrido. Torres et al. [TMG11] realizan un corte en una BVH y recorren en paralelo cada uno de los subárboles resultantes.

Otros autores proponen técnicas de empaquetado basadas directamente en las operaciones que los rayos demandan. El objetivo de estas propuestas es aprovechar al máximo las instrucciones SIMD. Trabajos como [WGBK07] y [GR08] se encuentran en esta categoría.

Por otro lado, el uso de un path tracing *unbiased* implica que no todos los rayos tienen la misma longitud. Para no desaprovechar recursos de la GPU, Novák et al. [NHD10] implementan un sistema que permite la regeneración de una ruta que ha terminado. Antwerpen [vA11] añade al trabajo anterior el uso de la coherencia de los rayos primarios mediante una compactación de los rayos regenerados.

Hermes et al. [HHGM10] y Hachisuka [Hac05] realizan

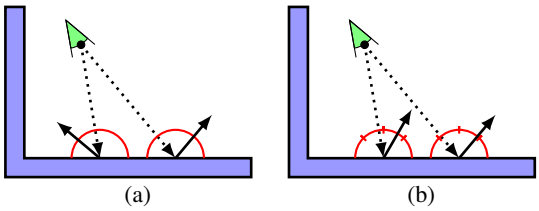


Figure 1: Esquema de generación coherente. Dos rayos (punteados) se generan en la cámara. En (a), los siguientes rayos se generan eligiendo un punto sobre todo el hemisferio. En (b), los siguientes rayos se generan coherentemente. Los hemisferios están divididos en patches y los dos rayos punteados pertenecen al mismo grupo. El siguiente rayo se genera dentro del mismo patch. Los rayos resultantes tienen direcciones parecidas.

un traversal de los rayos cuando todos tienen la misma dirección. Eso les capacita a usar el pipeline gráfico con proyección ortogonal para encontrar el punto de intersección más cercano a cada rayo.

3. Generación de rayos coherentes

Para elegir el siguiente rayo de una ruta, primero se calcula el espacio tangente en el punto de intersección usando la normal de la superficie y un vector no paralelo llamado *up*. Luego, se elige un punto sobre el hemisferio unidad y se transforma a coordenadas globales usando ese espacio tangente. Ese punto se usará para obtener la dirección del siguiente rayo de la ruta.

Si varios rayos tienen intersecciones cercanas y el mismo vector *up*, entonces un mismo punto del hemisferio va a ser transformado en puntos con coordenadas globales también cercanas. Así, las siguientes direcciones de cada ruta serán muy parecidas. Vamos a aprovechar esta propiedad en nuestra generación de rayos coherentes.

La generación coherente es un algoritmo que cambia la manera de elegir el siguiente rayo de una ruta sobre el hemisferio. En primer lugar, el conjunto de rayos se divide en grupos. Posteriormente, cada hemisferio sobre cada punto de intersección se divide en secciones llamadas *patches*. Por último, se selecciona aleatoriamente un patch para cada grupo de rayos. Los nuevos rayos van a ser elegidos como puntos aleatorios dentro de ese patch (Figura 1).

Los rayos primarios pueden ser fácilmente divididos en grupos de rayos coherentes usando los píxeles sobre los que se generan. Como los rayos de estos grupos tienen direcciones similares, entonces sus puntos de intersección van a estar próximos. Los rayos generados sobre estos puntos también van a ser coherentes ya que se generan sobre el mismo patch. Con ello hemos conseguido, con mucha probabilidad, que la coherencia se conserve en sucesivas reflexiones.

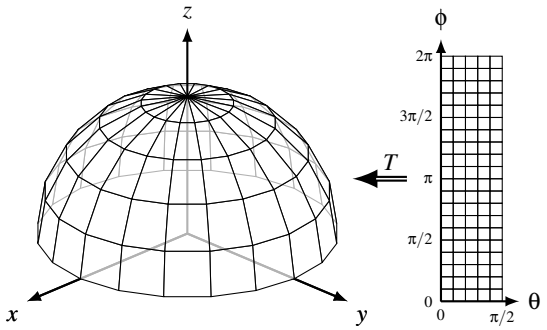


Figure 2: Patches del hemisferio con 5 divisiones de θ ($N = 5$) y 20 de ϕ ($M = 20$).

Para implementar la generación coherente, dividimos el hemisferio en $N \times M$ secciones, donde N y M son enteros. Consideremos la siguiente función T que transforma coordenadas esféricas en cartesianas

$$T(\theta,\phi) = (\sin\theta\cos\phi,\ \sin\theta\sin\phi,\ \cos\theta)$$

donde $\theta \in [0, \frac{\pi}{2}]$ y $\phi \in [0, 2\pi]$. Si dividimos el rango de la coordenada θ en N partes y el de la coordenada ϕ en M, el hemisferio queda dividido en $N \times M$ *patches* (Figura 2). Cada patch, por tanto, está definido como

$$Patch_{ij} = \{T(\theta,\phi) | \theta \in [i\Delta\theta, (i+1)\Delta\theta], \phi \in [j\Delta\phi, (j+1)\Delta\phi]\}$$

donde $\Delta\theta = \frac{\pi/2}{N}$, $\Delta\phi = \frac{2\pi}{M}$, $i \in \{0,1,\ldots,N-1\}$ y $j \in \{0,1,\ldots,M-1\}$. Es decir, un patch es la sección de hemisferio que genera T aplicada a cada rectángulo del espacio de coordenadas θ y ϕ. Por simplicidad, estableceremos $M = 4N$ durante el resto del artículo.

Las superficies *diffuse* son las que generan rayos más incoherentes, por tanto, a lo largo de este trabajo sólo vamos a considerar generación coherente en los puntos de intersección sobre superficies diffuse. Si algún rayo dentro de un grupo interseca con alguna superficie no diffuse, su siguiente rayo se genera según su BRDF, independientemente del resto y sin usar generación coherente.

Típicamente, el muestreo de un hemisferio para una superficie diffuse se lleva a cabo con una pdf proporcional al coseno del ángulo con la normal, lo que es conocido como *importance sampling*. Para hacer uso de esta característica vamos a hacer que la probabilidad de que el $Patch_{ij}$ sea elegido sea proporcional a dicho coseno. Para ello, se genera un punto o sobre todo el hemisferio con pdf $p(o) = \frac{(N \cdot o)}{\pi}$. El patch elegido es aquel que contiene al punto o. Posteriormente, se tiene que elegir un punto aleatorio dentro del patch. Esa elección se realiza con probabilidad uniforme dentro del propio patch.

4. Detalles de implementación

4.1. Descripción general

El renderizador usado es un path tracing implícito implementado en CUDA, usando una BVH construida con SAH como estructura de aceleración. La implementación de este algoritmo sigue las líneas de Antwerpen [vA11].

Para cada ruta es necesario guardar la información de su último rayo (origen y dirección) y el estimador parcial del color que aporta. Esta información se guarda en un array llamado *rays*. El orden de los rayos en este array coincide con el código Morton de los píxeles a los que pertenece. Sin embargo, los rayos van a ser reordenados durante el renderizado. Dicha reordenación no se realiza directamente sobre el array *rays*, sino usando un nivel más de indirección a través del array de enteros *idrays*.

Al comienzo del renderizado se lanza un hilo por cada píxel de la imagen. El id_g global de cada hilo representa el código Morton de las coordenadas de su píxel en la imagen. Cada hilo genera un rayo que sale de la cámara por dicho píxel id_g. Ese rayo se guarda en $rays[id_g]$ y su índice id_g en $idrays[id_g]$. Además, todos los hilos ponen a cero la imagen. Durante el renderizado, el array *idrays* va a ser reordenado (ver más abajo), por tanto, el valor $idrays[id_g]$ indicará el índice del rayo asociado a cada hilo.

La recursión de cada rayo termina en alguno de los siguientes casos: sale de la escena, encuentra una superficie luminosa o no es extendido debido a la ruleta rusa. En los dos primeros casos, la contribución de la ruta se acumula en la imagen. Si el rayo termina por ruleta rusa, su contribución es 0. En cualquiera de estos casos, el hilo vuelve a generar otro rayo desde la cámara por su píxel, regenerándose la ruta [NHD10].

Después de esta fase, los rayos regenerados se compactan al final del array y los extendidos quedan al principio. De esta manera, se intenta aprovechar la coherencia de los nuevos rayos primarios. Como ya se ha mencionado, esta ordenación no se hace directamente sobre el array de rayos *rays* sino sobre el array de índices *idrays*.

El algoritmo se ha implementado como un path tracing iterativo, de manera que se siguen extendiendo y regenerando rutas mientras no se haya alcanzado el tiempo de ejecución máximo. Sólo al principio del programa o cuando la cámara cambia de posición, el algoritmo tiene que comenzar de nuevo.

4.2. Kernels

El algoritmo completo se ha implementado en CUDA en cuatro kernels: *extends*, *compact*, *traversal* y *display*. Estos se ejecutan secuencialmente en este orden. Las imágenes renderizadas tienen una resolución de 1024×1024 y cada kernel se lanza con una configuración de rejilla de 2^{20} hilos (un hilo por rayo).

Extends genera el siguiente rayo de la ruta cuando se ha encontrado un punto de intersección. Si no se usa generación coherente, entonces el siguiente rayo se genera aleatoriamente usando la BRDF de la superficie pesada con el coseno. El caso de la generación coherente se explicará en la Subsección 4.3. Este kernel también se encarga de regenerar una ruta desde la cámara cuando la ruta ha terminado. *Compact* agrupa todos los rayos regenerados al final del array y los extendidos al principio. Está implementado con el *radixsort* de CUDPP 1.1.1 [HOS*10] usando sólo el bit menos significativo. *Traversal* encuentra el punto de intersección más cercano de cada rayo. Para este kernel se ha seguido el trabajo de Aila y Laine [AL09] que implementa el recorrido mediante *persistent threads*. *Display* muestra la información acumulada en la imagen parcial del path tracing. Su implementación se ha realizado para facilitar la interacción y depuración.

4.3. Generación coherente

Un grupo de rayos consiste en G hilos adyacentes, es decir, con índices globales id_g consecutivos. Por tanto, los hilos que pertenecen al mismo grupo tienen el mismo valor $\lfloor id_g/G \rfloor$. Los tamaños de G que hemos probado están relacionados con la forma en que se agrupan físicamente los hilos en CUDA. Concretamente, hemos considerado grupos con el tamaño de un warp ($G = 32$), un bloque ($G \in \{256, 512, 1024\}$) y el de toda la rejilla ($G = 2^{20}$).

Cuando el tamaño de G es un warp, un hilo del grupo se encarga de seleccionar el patch común al grupo. Posteriormente, el resto de hilos del warp recogen esa información de memoria compartida y cada uno prolonga su ruta hacia un punto sobre ese patch. No es necesario sincronización explícita por cómo se ejecutan los warps en GPU. Al path tracing que realiza la generación coherente de esta manera le hemos llamado `warp`. En esta configuración, el tamaño de bloque CUDA es de 256 hilos.

Cuando G es un bloque, un hilo se encarga también de elegir el patch común al grupo, pero esta vez es necesaria una sincronización explícita. Hemos usado la notación `block_256`, `block_512` y `block_1024` para referirnos a estos niveles de agrupación cuando $G = 256$, 512 y 1024, respectivamente.

Cuando G es la rejilla entera, todos los rayos pertenecen a un único grupo. En este caso es el *host* el que se encarga de elegir aleatoriamente el patch común a todos los rayos. Esto se implementa a través de una variable en la memoria global del dispositivo. El nombre que usaremos para este path tracing es `grid`.

Por último, hemos implementado también el path tracing tradicional en el que cada ruta se prolonga independientemente. A este algoritmo lo hemos llamado `normal` y lo usaremos como referencia.

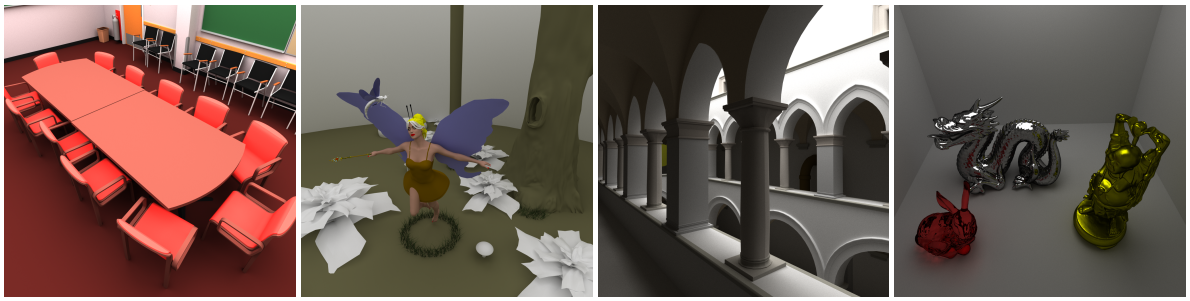


Figure 3: Imágenes de referencia de las escenas (ConfRoom, FairyForest, Sponza y Stanford) usadas en los tests.

4.4. Ruleta rusa por grupo

Después del kernel *traversal*, es posible que algunos rayos de un grupo tengan que regenerarse y otros extenderse. Si esto ocurre, los rayos extendidos van a ser compactados al principio del array. Así, la siguiente vez que se realice la generación coherente, los grupos van a estar formados por rayos generados sobre diferentes patches, perdiéndose parte de la propiedad de la coherencia.

De las tres causas de terminación de una ruta, la salida de un rayo fuera de la escena o la llegada a una superficie luminosa dependen tanto de la generación aleatoria de los rayos como de la escena, por tanto, no son controlables. Sin embargo, podemos implementar una ruleta rusa que no sea independiente por rayo. En este caso, el test de la ruleta rusa se realiza una sola vez para todo el grupo. Si se pasa este test, todos los rayos se extienden. Si no, todos terminan y se regeneran. La probabilidad de pasar el test es la misma que en el caso individual (ver Sección 5). En `warp` y `block`, sólo un hilo del grupo realiza la ruleta rusa en el kernel *extends*. En `grid`, el proceso es idéntico excepto que es el host el que decide si terminar todos los rayos.

Con este método de ruleta rusa por grupo, los estimadores siguen siendo *unbiased*. Además, el número de rayos consecutivos coherentes es más probable que no cambie, con lo que la compactación puede no ser necesaria. Veremos esto experimentalmente en la Sección 5. Los algoritmos que incorporan la ruleta rusa por grupo llevan el sufijo `_rr` y si no realizan el kernel *compact* entonces terminan en `(NO)`.

5. Resultados

Las escenas que hemos usado para los experimentos son *ConfRoom*, *FairyForest*, *Sponza* y *Stanford* (Figura 3). Las imágenes renderizadas tienen una resolución de 1024×1024 y se ha usado una cámara *pinhole*. En las tres primeras, los materiales de todas las superficies son diffuse. En la escena Stanford sólo lo son las paredes, mientras que el Buda tiene un material *glossy*, el Dragón, de espejo perfecto y el Conejo, de refracción perfecta. El *albedo* de todas las superficies está fijo a 0.8 y usamos este valor como probabilidad de continuación en todos los tests de ruleta rusa.

Con estas escenas probamos la generación coherente sobre rutas de diferentes características. Las escenas ConfRoom y FairyForest están muy bien iluminadas así que la mayoría de las rutas tienen longitud 2 (una reflexión). Para la primera, todo el techo de la habitación es un área de luz, mientras que la segunda está abierta por arriba. Por otra parte, Sponza tiene un área de luz encima del atrio y la cámara está enfocando un lugar donde parte de la iluminación es indirecta (más de una reflexión). Finalmente, Stanford tiene un área de luz en la parte superior de la escena y superficies no diffuse. En esta escena se mezclan superficies sobre las que se hace generación coherente (superficies diffuse) con las que no (resto de superficies).

Los algoritmos anteriormente descritos se han probado sobre una GeForce GTX 580 con 1.5 GB de DRAM. Todos los experimentos se han llevado a cabo renderizando la escena con un límite de 30 segundos para el tiempo acumulado de los kernels *extends*, *compact* y *traversal*. Se han probado diferentes valores de N entre 4 y 1000, mientras que se ha tomado la variable $M = 4N$ siempre.

Las gráficas de la Figura 4 muestran el número de rutas terminadas por píxel (de media). El algoritmo `normal` corresponde a una recta constante debido a que su ejecución no depende de N. Los algoritmos sin ruleta rusa ni compactación no se han incluido porque su rendimiento cae significativamente.

En primer lugar, se observa que el número de rutas terminadas con todos los algoritmos de generación coherente supera a `normal`. Se ha comprobado experimentalmente (usando *NVidia Visual Profiler*) que la etapa de traversal es más rápida debido a que las cachés tienen una menor tasa de fallos y a que el número de transferencias de memoria off-chip es menor. En segundo lugar, según crece N, las direcciones generadas se hacen más parecidas dentro de cada grupo porque los patches son más pequeños. En consecuencia, las curvas de nuestros algoritmos son crecientes, es decir, el traversal es más rápido según crece N.

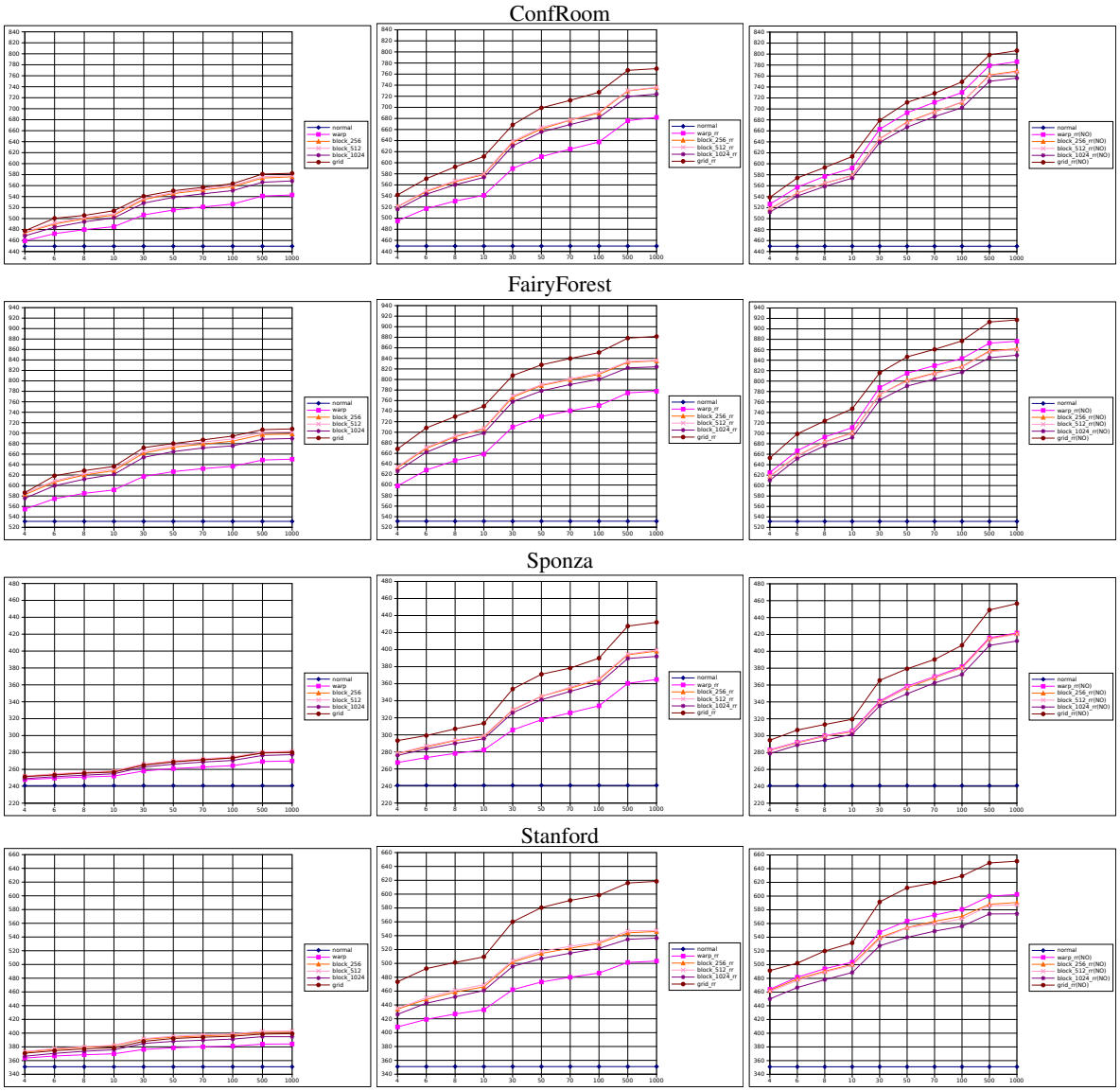


Figure 4: Rutas terminadas por píxel en media.

Los algoritmos que generan a nivel de grid son los que ofrecen mejor rendimiento debido a que su tasa de aciertos de caché es la mayor. Después se encuentran los algoritmos basados en bloque y en warp. En las tres columnas, las curvas de los algoritmos que generan en bloque son muy parecidas. Las curvas correspondientes a los tamaños 256 y 512 discurren juntas, y la asociada al tamaño 1024 queda ligeramente por debajo. El motivo es que con bloques de 1024 hilos, sólo un bloque se asigna a cada multiprocesador de la GPU, por lo que las barreras de sincronización suponen una penalización mayor que para los otros tamaños.

En cuanto a las curvas correspondientes a la agrupación a nivel de warp, su ubicación depende de la columna en cuestión. En las dos primeras columnas ofrecen el peor rendimiento mientras que en la tercera quedan en segundo lugar, superando a los algoritmos por bloque. Recuérdese que la única diferencia entre las columnas segunda y tercera consiste en que en la tercera no se compacta. Así el beneficio obtenido cuando se elimina la compactación es mayor para la agrupación en warps que para la agrupación en bloques. La explicación es que los grupos pequeños sufren más que los grandes tras la compactación. Si, por ejemplo, uno de los primeros rayos decide regenerarse, entonces se coloca al final del array *idray*. Esto supone desorganizar los

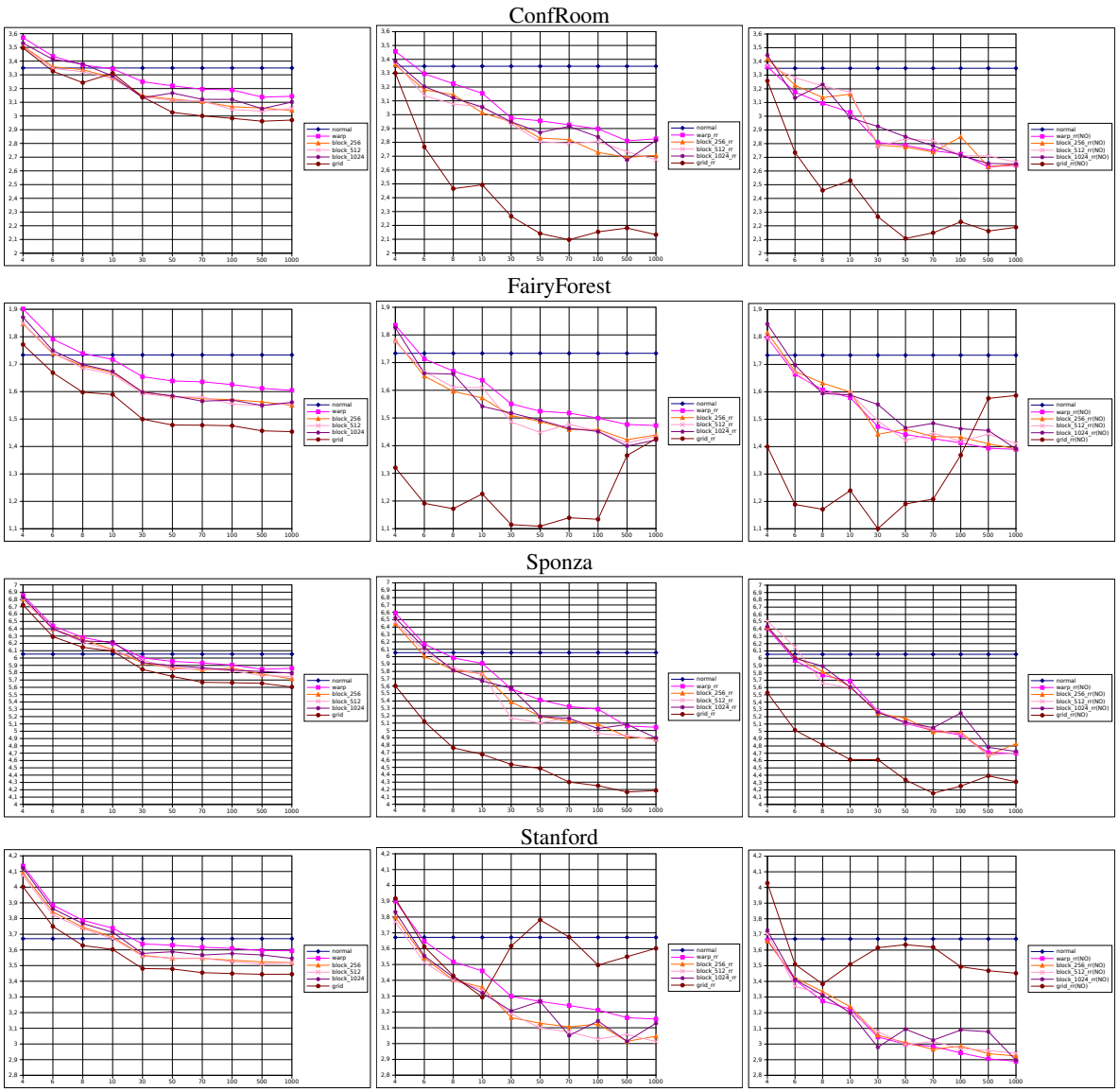


Figure 5: RMS (en %) de los algoritmos con respecto a las imágenes de referencia de la Figura 3.

actuales grupos de rayos, ya que pierden uno de sus rayos e introducen otro del siguiente grupo. Como cada grupo se ejecuta en la GPU en warps de 32 hilos (ejecución en SIMT) esta reorganización afecta más al agrupamiento basado en warp porque todos los grupos han visto modificada la composición de sus rayos.

En la Figura 5 se analiza el error de las imágenes resultantes medido como la raíz del error cuadrado medio (*root mean square* o *RMS*) de cada píxel (canales RGB) con respecto a las imágenes de referencia (Figura 3). Estas imágenes se han obtenido aplicando el algoritmo `normal` durante 20 minutos.

Cuando N crece, el número de rutas terminadas aumenta, lo que supondría mayor calidad en la imagen. Pero al mismo tiempo la correlación entre las muestras de los píxeles del mismo grupo también será mayor, lo que implicaría un mayor error. Las curvas de la Figura 5 indican que tiene mayor influencia la ganancia que aporta un número mayor de rutas terminadas, ya que se trata de curvas decrecientes en general. De hecho, comienzan con más error a pesar de que el número de rutas terminadas es mayor. A medida que el número de rutas terminadas crece, el error cae por debajo de `normal`. Sin embargo, los algoritmos basados en grid tienen un comportamiento más imprevisible, como se

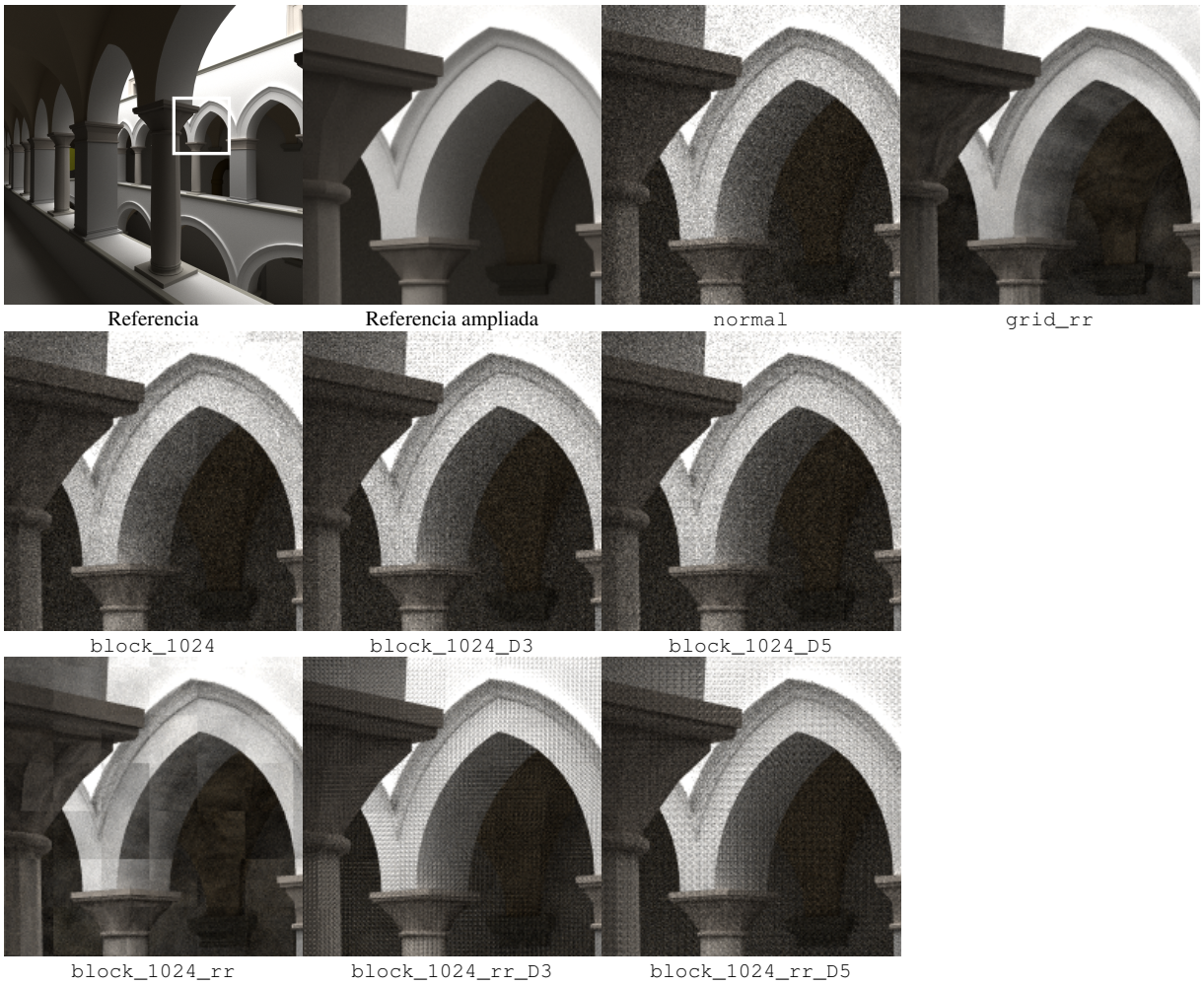


Figure 6: Detalles de algunas capturas de la escena Sponza. Las imágenes han sido obtenidas con $N = 100$. Todas las imágenes tienen un RMS menos que `normal`.

puede ver en las gráficas de la segunda y tercera columna en FairyForest y Stanford.

En la Figura 6 presentamos capturas de las imágenes generadas por algunos de nuestros algoritmos. La influencia de la correlación entre las muestras de los píxeles del mismo grupo aparece visualmente como un patrón de rejilla cuadrada para los algoritmos basados en bloque. Para los basados en warp sucede algo parecido aunque no lo mostramos por falta de espacio. Este efecto se hace especialmente visible para valores grandes de N y con ruleta rusa por grupo. Aunque el RMS es menor que el del algoritmo `normal`, este patrón resulta molesto desde un punto de vista subjetivo.

Para aliviar su efecto, hemos usado una técnica relacionada con el *interleaved sampling* [KH01]. Consiste en modificar las coordenadas del píxel al que cada ruta aporta, multiplicando cada coordenada por un entero impar D. Los valores que hemos usado para nuestros experimentos son $D = 3$ y $D = 5$. Esta técnica tiene la ventaja de que substituye el patrón de rejilla por un ruido que se reparte por la imagen. Sin embargo, su uso implica que los rayos primarios van a tener menos coherencia ya que son generados en píxeles más alejados entre sí. Esto se aprecia experimentalmente como una disminución del número de rutas terminadas. En la Figura 7, se muestra el RMS usando $D = 5$. Se aprecia que el error de los algoritmos que usan interleaved sampling (con sufijo `D5`) es mayor que los que no lo usan. Los algoritmos de la primera columna son los más lentos y su error no supera al error de `normal`. En las otras columnas, los algoritmos también tienen un error mayor, pero sí superan a `normal` a medida que crece N.

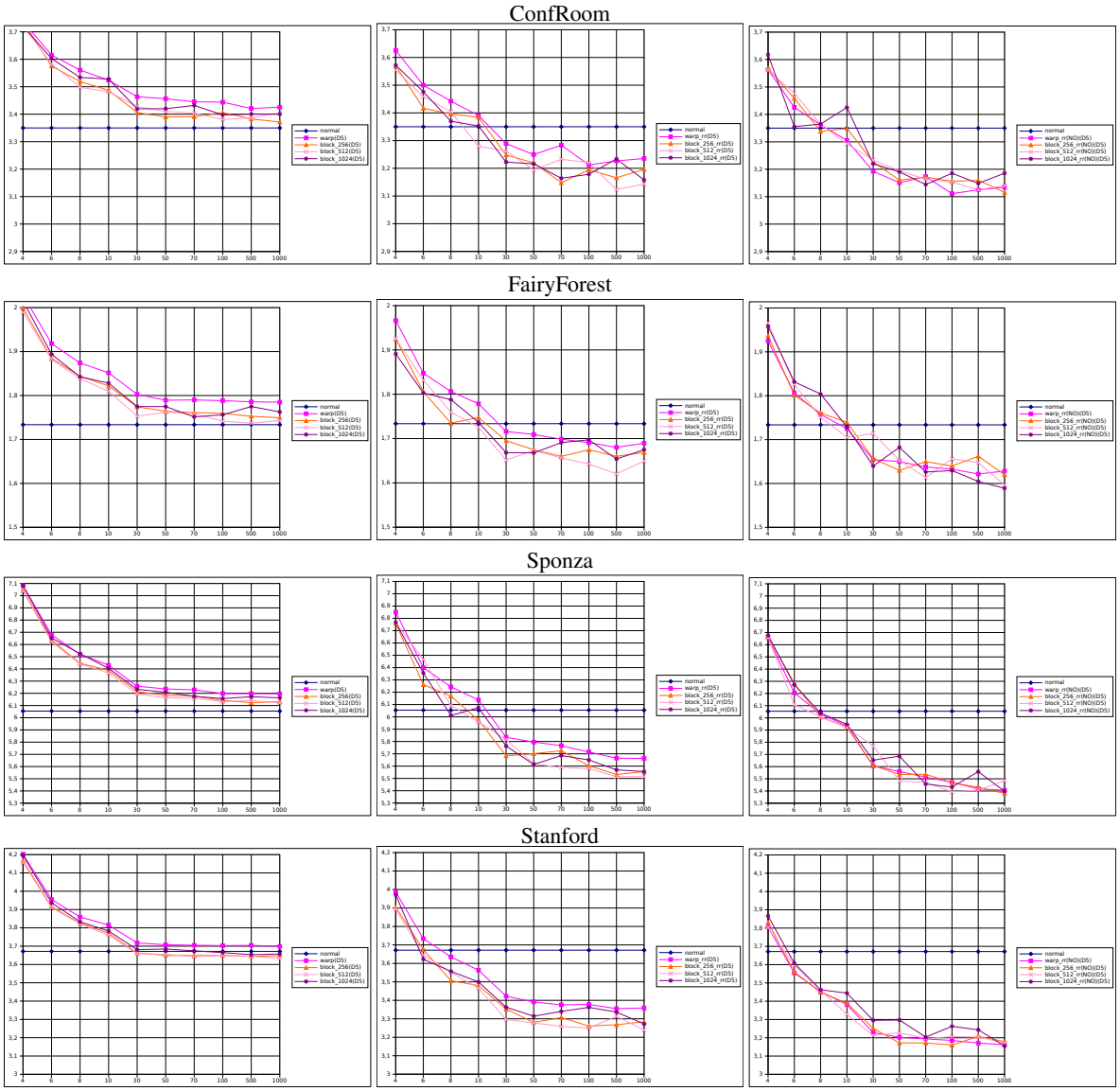


Figure 7: RMS (en %) de los algoritmos con respecto a las imágenes de referencia con D=5.

6. Trabajo futuro

Los patches en los que está dividido el hemisferio no poseen la misma área. Habría que comprobar la técnica de generación coherente cuando los patches son todos iguales (como en las esferas geodésicas) o cuando se genera por una función que preserva áreas (como en [SC97]).

La generación coherente se puede aplicar siempre que se tenga que muestrear un conjunto de direcciones. En este sentido, se podría usar sobre superficies con BRDFs no lambertianas o con medios participativos.

El rendimiento se puede degradar en cada reflexión de las rutas. Eso se debe a que los puntos de intersección están cada vez más alejados entre sí, intersecan superficies con normales muy diferentes o algún rayo del grupo tiene que terminar. Para aliviar estos problemas se proponen dos soluciones. La primera es implementar una fase de reordenación de rayos en función de sus cualidades geométricas. Esta fase se podría realizar sólo cada cierto número de reflexiones para que no sea el cuello de botella del rendimiento. La segunda es la terminación de todos los rayos del grupo siempre que alguno tuviera que terminar. En este trabajo se ha realizado esto en la ruleta rusa por grupo, pero se podría extender a los otros dos casos de terminación.

7. Conclusiones

En este trabajo hemos planteado una manera diferente de extender rutas en path tracing llamada *generación coherente*. Esta generación es fácil de implementar y aprovecha mejor la forma de computación de las GPUs. En concreto, el conjunto de rayos es dividido en grupos. Para determinar la extensión de cada ruta, el hemisferio sobre cada punto de intersección se divide en patches y cada grupo elige uno de ellos. El siguiente rayo de cada ruta se genera aleatoriamente sobre ese patch. Con ello se consigue que la coherencia del grupo se conserve a lo largo del trazado de las rutas de cada grupo, acelerando así la fase de traversal. En consecuencia, se terminan más rutas y las imágenes obtenidas, en un mismo período de tiempo, son de mayor calidad.

Sobre esta idea, hemos implementado en CUDA distintas configuraciones de un path tracing. Distinguimos casos por la forma en que se agrupan los rayos, y el número de divisiones del hemisferio, así como por la forma de organización del algoritmo de renderizado, incluyendo o no técnicas como la compactación y la ruleta rusa. Concluimos que, en general, nuestras propuestas tienen un rendimiento mejor que el algoritmo clásico basado en la generación puramente aleatoria de rayos.

References

[AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *High-Performance Graphics* (2009), pp. 145–149. 2, 4

[BEL*07] BOULOS S., EDWARDS D., LACEWELL J. D., KNISS J., KAUTZ J., WALD I., SHIRLEY P.: Packet-based Whitted and Distribution Ray Tracing. In *Graphics Interface* (2007), pp. 177–184. 2

[BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive Ray Packet Reordering. *Symposium on Interactive Ray Tracing* (2008), 131–138. 2

[GL10] GARANZHA K., LOOP C.: Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. In *Eurographics* (2010). 2

[GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Interactive Ray Tracing* (2007), pp. 113–118. 2

[GR08] GRIBBLE C. P., RAMANI K.: Coherent Ray Tracing via Stream Filtering. In *Eurographics Symposium on Interactive Ray Tracing* (2008), pp. 59–66. 2

[Hac05] HACHISUKA T.: High-Quality Global Illumination Rendering Using Rasterization. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 615–633. 2

[HHGM10] HERMES J., HENRICH N., GROSCH T., MUELLER S.: Global Illumination using Parallel Global Ray Bundles. In *Vision, Modeling and Visualization* (2010), pp. 65–72. 2

[HOS*10] HARRIS M., OWENS J. D., SENGUPTA S., TSENG S., ZHANG Y., DAVIDSON A., SATISH N.: CUDA Data Parallel Primitives Library (CUDPP 1.1.1), 29 April 2010. http://code.google.com/p/cudpp/. 4

[HSMH07] HORN D. R., SUGERMAN J., MIKE H., HANRAHAN P.: Interactive KD-Tree GPU Raytracing. In *I3D* (2007), pp. 167–174. 2

[Jen96] JENSEN H. W.: Global Illumination Using Photon Maps. In *Eurographics Workshop on Rendering Techniques* (1996), pp. 21–30. 2

[Kaj86] KAJIYA J. T.: The Rendering Equation. *SIGGRAPH Computer Graphics 20*, 4 (1986), 143–150. 1

[Kel97] KELLER A.: Instant Radiosity. In *Computer Graphics and Interactive Techniques* (1997), pp. 49–56. 2

[KH01] KELLER A., HEIDRICH W.: Interleaved Sampling. In *Eurographics Workshop on Rendering* (2001). 8

[MMAM07] MANSSON E., MUNKBERG J., AKENINE-MOLLER T.: Deep Coherent Ray Tracing. In *Symposium on Interactive Ray Tracing* (2007), pp. 79–85. 2

[NFLM07] NAVRATIL P. A., FUSSELL D. S., LIN C., MARK W. R.: Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization. In *Symposium on Interactive Ray Tracing* (2007), pp. 95–104. 2

[NHD10] NOVÁK J., HAVRAN V., DASCHBACHER C.: Path Regeneration for Interactive Path Tracing. In *Eurographics, short papers* (2010), pp. 61–64. 2, 4

[NUG09] NOGUERA J. M., UREÑA C., GARCÍA R. J.: A Vectorized Traversal Algorithm for Ray-Tracing. In *GRAPP* (2009), pp. 58–63. 2

[NVI11] NVIDIA: NVidia CUDA C Programming Guide 4.1 www.nvidia.com, 2011. 2

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum 26*, 3 (2007), 415–424. 2

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering Complex Scenes with Memory-Coherent Ray Tracing. In *SIGGRAPH* (1997), pp. 101–108. 2

[SC97] SHIRLEY P., CHIU K.: A Low Distortion Map Between Disk and Square. *J. Graphics Tools 2*, 3 (1997), 45–52. 9

[TMG11] TORRES R., MARTIN P., GAVILANES A.: Traversing a BVH Cut to Exploit Ray Coherence. In *GRAPP* (2011), pp. 140–150. 2

[vA11] VAN ANTWERPEN D.: Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU. In *High Performance Graphics* (2011), pp. 41–50. 2, 4

[Vea98] VEACH E.: *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, 1998. 1

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive Rendering with Coherent Ray Tracing. In *Computer Graphics Forum (Proceedings of Eurographics'01)* (2001), vol. 20, pp. 153–164. 2

[WGBK07] WALD I., GRIBBLE C. P., BOULOS S., KENSLER A.: *SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering*. Tech. rep., 2007. 2

[WKB*02] WALD I., KOLLIG T., BENTHIN C., KELLER A., SLUSALLEK P.: Interactive Global Illumination Using Fast Ray Tracing. In *Eurographics Workshop on Rendering* (2002), pp. 15–24. 2